Erik Wilde
Cesare Pautasso *Editors*

REST: From Research to Practice

Springer

REST: From Research to Practice

Erik Wilde • Cesare Pautasso
Editors

REST: From Research to Practice

Springer

Editors
Erik Wilde
School of Information
UC Berkeley
Berkeley, CA
USA
dret@berkeley.edu

Cesare Pautasso
Faculty of Informatics
University of Lugano
Via Buffi 13
6900 Lugano
Switzerland
c.pautasso@ieee.org

ISBN 978-1-4419-8302-2 e-ISBN 978-1-4419-8303-9
DOI 10.1007/978-1-4419-8303-9
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2011932174

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Contents

Contributors

Paul Adamczyk Booz Allen Hamilton Inc., paul.adamczyk@gmail.com

Erik Albert Department of Computer Science, University of Maine, 237 Neville Hall, Orono, ME 04469-5752, USA, albert17@cs.umaine.edu

Jan Algermissen NORD Software Consulting, Kriemhildstrasse 7, 22559 Hamburg, Germany, algermissen@acm.org

Ilkay Altintas San Diego Supercomputer Center, University of California, San Diego, La Jolla, CA 92093, USA, altintas@sdsc.edu

Mike Amundsen Erlanger, KY 41018, USA, mamund@yahoo.com

Mauro Caporuscio Politecnico di Milano, Piazza Leonardo, Da Vinci 32, 20133 Milano, Italy, caporuscio@elet.polimi.it

Sudarshan S. Chawathe Department of Computer Science, University of Maine, 237 Neville Hall, Orono, ME 04469-5752, USA, chaw@cs.umaine.edu

Chris Churas Center for Research in Biological Systems, University of California, San Diego, La Jolla, CA 92093, USA, churas@ncmir.ucsd.edu

Duncan Cragg ThoughtWorks (UK) Ltd., Berkshire House, 168–173 High Holborn, London, WC1V 7AA, restbook@cilux.org

Antonio Cuadra-Sánchez Indra Sistemas, Parque Tecnológico de Boecillo, 47151 Valladolid, Spain, acuadra@indra.es

María del Mar Cutanda-Rodríguez Indra Sistemas, C/Anabel Segura 7, 28108 Alcobendas (Madrid), Spain, mdcutanda@indra.es

Frank Eliassen Department of Informatics, University of Oslo, PO Box 1080 Blindern, 0316 Oslo, Norway, frank@ifi.uio.no

Mark Ellisman Center for Research in Biological Systems, University of California, San Diego, La Jolla, CA 92093, USA, mark@ncmir.ucsd.edu

Federico Fernandez Department of Computer Science, Universidad Catolica de Chile, Santiago, Chile

Jordi Fernández Esilog Consulting, S.L., Aribau 112, Barcelona, Spain, jordi.fernandez@esilog.com

Damaris Fuentes-Lorenzo Carlos III University, Av. de la Universidad 30, 28911 Madrid, Spain, dfuentes@it.uc3m.es

Marco Funaro Politecnico di Milano, Piazza Leonardo, Da Vinci 32, 20133 Milano, Italy, funaro@elet.polimi.it

Antonio Garrote Hernández University of Salamanca, Avenida Italia 29 4-A, Plaza de los Caídos, s/n, 37008, Salamanca, Spain, agarrote@usal.es, antoniogarrote@gmail.com

Carlo Ghezzi Politecnico di Milano, Piazza Leonardo, Da Vinci 32, 20133 Milano, Italy, ghezzi@elet.polimi.it

Jeff Grethe Center for Research in Biological Systems, University of California, San Diego, La Jolla, CA 92093, USA, jgrethe@ncmir.ucsd.edu

Dominique Guinard Institute for Pervasive Computing, ETH Zurich, Switzerland, dguinard@guinard.org

Madhusudan Gujral San Diego Supercomputer Center, University of California, San Diego, La Jolla, CA 92093, USA, madhu@sdsc.edu

Munawar Hafiz University of Illinois at Urbana-Champaign, 201 N Goodwin Avenue, Urbana, IL 61801, USA, mhafiz@illinois.edu

Lars Hagge Deutsches Elektronen-Synchrotron, Notkestrasse 85, Hamburg 22607, Germany, lars.hagge@desy.de

Michael Hausenblas DERI, National University of Ireland Galway, IDA Business Park, Galway, Ireland, michael.hausenblas@deri.org

Ralph E. Johnson University of Illinois at Urbana-Champaign, 201 N Goodwin Avenue, Urbana, IL 61801, USA, rjohnson@illinois.edu

Jacek Kopecký Knowledge Media Institute, Open University, Walton Hall, Milton Keynes, MK7 6AA, UK, j.kopecky@open.ac.uk

Abel W. Lin Center for Research in Biological Systems, University of California, San Diego, La Jolla, CA 92093, USA, awlin@ncmir.ucsd.edu

Benoit Macq Laboratoire de Télécommunications et Télédétection – TELE, Université catholique de Louvain, Louvain-la-Neuve, Belgium, benoit.macq@uclouvain.be

Maria Maleshkova Knowledge Media Institute, Open University, Walton Hall, Milton Keynes, MK7 6AA, UK, m.maleshkova@open.ac.uk

Simon Mayer Distributed Systems Group, Institute for Pervasive Computing, ETH Zurich, CNB, Universitätstrasse 6, 8092 Zurich, Switzerland, simon.mayer@inf.ethz.ch

Hildeberto Mendonça Laboratoire de Télécommunications et Télédétection – TELE, Université catholique de Louvain, Louvain-la-Neuve, Belgium, me@hildeberto.com

María N. Moreno García University of Salamanca, Plaza de los Caidos, s/n, 37008, Salamanca, Spain, mmg@usal.es

Mathias Mueller Software Engineering Group, University of Fribourg, Switzerland

Jaime Navon Department of Computer Science, Universidad Catolica de Chile, Santiago, Chile, jnavon@ing.puc.cl

Vincent Nicolas UCL/TELE, Universite catholique de Louvain, Batiment Stevin, 1er etage 2, Place du Levant, 1348 Louvain-la-Neuve, Belgique, vincent.nicolas@uclouvain.be

Guy Pardon ATOMIKOS, Hoveniersstraat 39/1, 2800, Mechelen, Belgium, guy@atomikos.com

Cesare Pautasso Faculty of Informatics, University of Lugano, via Buffi 13, 6900 Lugano, Switzerland, c.pautasso@ieee.org

Carlos Pedrinaci Knowledge Media Institute, Open University, Walton Hall, Milton Keynes, MK7 6AA, UK, c.pedrinaci@open.ac.uk

Ivan Porres Department of Information Technologies ICT, Abo Akademi University, Joukahainengatan 3-5 A, FI-20520, ABO, Finland

Irum Rauf Department of Information Technologies ICT, Abo Akademi University, Joukahainengatan 3-5 A, FI-20520 ABO, Finland, irauf@abo.fi

Ian Robinson Neo Technology, Menlo Park, CA, USA, iansrobinson@gmail.com

Javier Rodriguez Esilog Consulting, S.L., Calle Aribau 112, 2° 2ª, 08036 Barcelona, Spain, javier@rodriguez.org.mx, javier.rodriguez@esilog.com

Daniel Romero INRIA Lille – Nord Europe, Parc Scientifique de la Haute Borne, 40, avenue Halley – Bât. A, Park Plaza, 59650 Villeneuve d'Ascq, France, daniel.romero@inria.fr

Romain Rouvoy INRIA Lille, University of Lille 1, INRIA Lille – Nord Europe, Parc Scientifique de la Haute Borne, 40, avenue Halley – Bât. A, Park Plaza, 59650 Villeneuve d'Ascq, France, romain.rouvoy@lifl.fr

Luis Sánchez Univeristy Carlos III of Madrid, Av. de la Universidad 30, 28911 Leganés (Madrid), Spain, luiss@it.uc3m.es

Juha Savolainen Nokia Research Center, Visiokatu 1, Tampere 33720, Finland, juha.e.savolainen@nokia.com

Petri Selonen Nokia Research Center, Visiokatu 1, Tampere 33720, Finland, petri.selonen@nokia.com

Patrick H. Smith Booz Allen Hamilton Inc., patrick.h.smith@gmail.com

Vlad Stirbu Nokia Research Center, Visiokatu 1, Tampere 33720, Finland, vlad.stirbu@nokia.com

Daniel Szepielak Deutsches Elektronen-Synchrotron, Notkestrasse 85, Hamburg 22607, Germany, daniel.szepielak@desy.de

Amirhosein Taherkordi Department of Informatics, University of Oslo, PO Box 1080 Blindern, 0316 Oslo, Norway, amirhost@ifi.uio.no

Vlad Trifa Institute for Pervasive Computing, ETH Zurich, Universitätstrasse 6, 8092 Zurich, Switzerland, trifa@acm.org

Przemyslaw Tumidajewicz Deutsches Elektronen-Synchrotron, Notkestrasse 85, Hamburg 22607, Germany, przemyslaw.tumidajewicz@desy.de

Tomas Vitvar Institut für Informatik, University of Innsbruck, Technikerstrasse 21a, 6020 Innsbruck, Austria, tomas@vitvar.com

Olga Vybornova Laboratoire de Télécommunications et Télédétection – TELE, Université catholique de Louvain, Louvain-la-Neuve, Belgium, olga.vybornova@uclouvain.be

Erik Wilde School of Information, UC Berkeley, Berkeley, CA, USA, dret@berkeley.edu

Part I
Foundations

Introduction

Cesare Pautasso and Erik Wilde

Web Services

Anybody following the discussions around "Web Services" in recent years is aware of the fuzzy definition of the term, and a little bit of history can quite easily explain some of the confusions around current terminology (or use of terminology). The general idea of using Web technologies to not only deliver Web pages (HTML documents) between HTTP clients and servers appeared probably more than 10 years ago, when it became clear that the Web and its technical foundations of URIs, HTTP, and HTML were becoming a very widely deployed information delivery and service platform. Late in the 1990s, one major approach of implementing this idea gained a lot of traction, the *Simple Object Access Protocol (SOAP)* (Box et al. 1999). SOAP used the new *Extensible Markup Language (XML)* (Bray et al. 1998) as a packaging format for a *Remote Procedure Call (RPC)* mechanism, and thus simply used the well-established pattern of using RPC mechanisms for implementing distribution, and packaged it using the Web technologies XML and HTTP. Before that, most RPC mechanisms used their own packaging/marshalling formats, and oftentimes even their own delivery protocols, so reusing existing Web technologies for this made sense, and reduced the amount of proprietary technologies required for RPC implementations. Additionally, tunneling SOAP messages through HTTP had the great advantage of using a protocol which would – by default – go through corporate firewalls and thus greatly facilitate the integration of distributed applications in business to business scenarios.

While SOAP as a Web implementation of the RPC concept gained a lot of traction and for a while was synonymous with what people meant when they referred to "Web Services", it soon became clear that SOAP, while using Web technologies for transporting RPC calls, did not really implement a model of "Web Services" that

C. Pautasso (✉)
Faculty of Informatics, University of Lugano, via Buffi 13, 6900 Lugano, Switzerland
e-mail: c.pautasso@ieee.org

took the architectural principles of the Web into account (Vinoski 2008a). While questions around SOAP's ability to implement true "Web Services" (instead of just implementing RPC over the Web) surfaced relatively early (Prescod 2002), SOAP had already gained considerable momentum and most major vendors had joined the standardization process of SOAP and related technologies. *Representational State Transfer (REST)* (Fielding 2000) as a post-hoc conceptualization of the Web as a loosely coupled decentralized hypermedia system was coined as a term in 2000, but it took several years until it became clearly visible in the mainstream that the model of "Web Services are based on SOAP" had a serious competitor in the form of services that better conformed to the architectural principles of the Web. Because those principles were defined by the REST model, this new variety of Web Services often was referred to as "RESTful Web Services."

At the time of writing, it is probably safe to say that most people will ask when somebody refers to "Web Services" to make sure whether they refer to the RPC model, the REST model, or maybe a more generic and vague concept of any kind of service delivered using Web technologies. While we cannot change this general confusion or just fundamental vagueness of this term, it is important to understand that the most important difference between the two "flavors" of Web services is the architectural starting point, not so much the actual choice of technologies. Nowadays, instead of referring to SOAP, oftentimes this flavor of Web services is referred to as "WS-*" Web services, referring to the multitude of WS-prefixed middleware interoperability standards that were developed over the years to add expressivity to the basic SOAP format. Since the main differences are architectural and not on the level of technology choices, it is important to focus on this level when comparing these approaches, and several attempts have been made to compare them as objectively as possible (Pautasso et al. 2008).

It is not possible to simply say that one variety is better than the other, but since RESTful Web services gained momentum, it has become clear that they do provide certain advantages in terms of simplicity, loose coupling (Pautasso and Wilde 2009), interoperability, scalability and serendipitous reuse (Vinoski 2008b) that are not provided to the same degree by WS-*.

REST Definition

Simply speaking, REST is a set of constraints that inform the design of an hypermedia system. The claim of REST is that following those constraints will result in an architecture that works well in the areas of scalability, mashup-ability, usability, and accessibility. Like all claims on this level of abstraction, this is not really something that can be proven, but it seems to be accepted nowadays that particularly in areas where there is no centralized coordination of the design of all initial and future components of an information system, REST indeed does lead to designs that are less tightly coupled than the more established architectures that have been informing the design of distributed systems and enterprise IT architectures.

The following constraints can be considered as being the core of the REST architectural style:

1. *Resource Identification*: All resources that are relevant for an application (and its state) should be given unique and stable identifiers. These identifiers should be global, so that they can be dereferenced independent of context. It is important that the concept of a "resource" in this case is not limited to the static "things" that an application is dealing with; it also comprises all information that is required to talk *about* those things, such as [transactional documents such as orders].
2. *Uniform Interface*: All interactions should be built around a uniform interface, which supports all the interactions with resources by providing a general and functionally sufficient set of methods. This constraint is in stark contrast to RPC, where the main facility for exposing functionality is to define a set of methods that can be invoked remotely, whereas in REST, there is no such this as "methods" that can be "called." Instead, RESTful services expose resources and resource interactions can only use the uniform interface, or a subset of it.
3. *Self-Describing Messages*: For the interactions with resources through the uniform interface, REST demands to use resource representations that represent the important aspects of the resources. Those representations have to be designed in a way that participating parties can get a complete understanding of resources or relevant state by just inspecting representations. Changes of resource or state also are communicated by exchanging representations through the uniform interface. In order to support this constraint, the uniform interface must provide a way in which information exchanges can "label" representations, so that no out-of-band information or prior agreement is necessary to "understand" a representations that is received. It is important to understand that "self-describing" in this case does not refer to the term as it is sometimes used in the context of semantics, but only refers to the fact that in order to be able to process a representation that is exchanged through the uniform interface, no out-of-band information is required.
4. *Hypermedia Driving Application State*: The representations that are exchanged are supposed to be linked, so that an application that understands a representation will be able to find the links, will understand them because their semantics are defined by the representation, and will be able to use them because they lead to other identified resources that can be interacted with through the uniform interface. Without links, it would be impossible to expose new resources or to provide applications with the possibility to make certain state transitions, and the hypermedia constraint is probably the one that is most important for supporting loose coupling (Pautasso and Wilde 2009), because identifiers can be discovered at runtime and interacted with through the uniform interface, without the need of any additional previous agreements between interacting parties.
5. *Stateless Interactions*: This constraint means that each interaction between a client and a server has to be entirely self-contained; there should be no client state (often referred to as a "session") maintained on the server which would allow an interaction to depend on both the exchanged representation and on the

session associated with the client. Any interaction can, of course, cause a change in a resource, in which case the next interaction with that resource will reflect that changed resource state. But this change in *resource state* is different of a server-maintained *client session*, because the server only needs to keep track of resources states, but not of client sessions. This constraint is important to ensure that the scalability of servers is bound only by the number of concurrent client requests and not by the total number of clients that they have to interact with.

The general claim of RESTful systems implementing these constraints are that they are highly scalable and that the interlinking of self-describing representation formats allows such a system to grow organically and in a decentralized way. The Web is a very impressive demonstration of a system that does implement those constraints, and in those places where Web components have violated those constraints (such as the infamous "session objects" in various Web-oriented frameworks), important issues such as scalability indeed suffered – and often this was only discovered when an implementation was almost complete or already deployed and the server load grew past a critical point.

REST Maturity Models

The main constraints of REST as introduced in the previous section can be regarded as checkpoints to judge whether a given design is indeed RESTful or not. REST has become popular enough so that many simply perceive it as a label saying "this works well on the Web," and many APIs and services that label themselves as being RESTful are not.

One popular model for analyzing services has been dubbed the "Richardson Maturity Model," named after Leonard Richardson.[1] It distinguishes four levels, named "0" to "3," and categorizes services according to their adherence to REST constraints.

- *Level 0* are services that simply exchange XML documents over HTTP, such as XML-RPC. In this case, there is no REST at all, and the only reason why some of these services my label themselves as REST is because they are not using WS-* standards. They still (mis)use HTTP as a tunneling protocol.
- *Level 1* are services that use resource identification and build interactions on top of these identified resources. In this case, at least the managed resources are exposed as identifiable resources so that they can be directly addressed by clients. However, in most cases resource URIs of Level 1 services correspond to method

[1]A similar but more refined classification of HTTP-based APIs was developed by Jan Algermissen in `http://nordsc.com/ext/classification_of_http_based_apis.html`. In particular, he adds the facet of whether self-describing messages are being used, in the sense that they have to be explicitly labeled with a media type.

identifiers and are also used to pass parameters making only limited use of the HTTP expressive power.

- *Level 2* means that in addition to fine-grained resource addressing, also HTTP methods are properly used as intended by the REST uniform interface. Resource interactions are thus designed in a way that maps well to this constraint. This not only means that HTTP's methods are properly used, it also means that HTTP's status codes are used to indicate the correct result of applying a method to a resource. Since HTTP methods are used properly, HTTP's classification of methods as safe and/or idempotent can be used to optimize the system using intermediaries.
- *Level 3* adds hypermedia controls to resource representations, so that clients can interact with resources by simply following links. Those links in most cases will need to be typed so that clients can understand the semantics of a link, but the important issue is that clients can now *explore* an open space of resources, instead of having to know everything in advance.

Generally speaking, these attempts at providing a simple framework to decide "how RESTful" a given service is demonstrates that the current landscape of services that claim to be RESTful is in need of closer analysis (Maleshkova et al. 2010). REST as a design principle claims to create designs that have positive properties, but it is unlikely that these properties can be expected with designs that ignore certain key constraints.

The current landscape of REST design methods and implementation platforms is still in development, and it remains to be seen whether research efforts will make it easier to both design and implement systems that are truly RESTful, and to test systems for their design qualities in a systematic way.

Describing RESTful Services

One of the core components of the early SOAP-oriented approach to Web services was the idea of a *service directory*, so that services could be located by using that directory, and in that directory service descriptions would make it possible to both understand what a service is about, and how it has to be used. In the world of SOAP and WS-*, service description is done by using the *Web Services Description Language (WSDL)* (Chinnici et al. 2007), and the most popular approach for managing those descriptions in a directory is to use *Universal Description Discovery and Integration (UDDI)* (Clement et al. 2004).

One of the important reasons why service description and discovery is very important in the WS-* approach is that every service exposes a specific interface, and without a description of that interface, it is impossible to use that service. REST's constraint of a *uniform interface*, on the one hand, removes the need for a specific description of a service's interface, and the constraint of *hypermedia driving application state* removes the need for specific discovery of services, because they are discovered by simply following links.

Nevertheless, describing services can be useful, even though it may not be strictly necessary. One important reason can be to describe a service for documentation, so that users of that service know what to expect. There is the risk of such a description and documentation being outdated by newer versions of the service, so clients of that service should always rely on the actual REST mechanisms (using the uniform interface, dynamically negotiating the actual representation format, and following links they find in self-describing messages). However, having an explicit documentation can be helpful and can be a good way to explain a service in a more abstract way than just using it, and thus there have been various proposals on how to describe RESTful services.

One of the most popular approaches is the *Web Application Description Language (WADL)* (Hadley 2006). WADL's main weakness is the lack of support for the hypermedia nature of RESTful services; it described resources based on URI path structures, and thus tightly couples a WADL-based client to a fixed scheme of how URIs are used. The *Resource Linking Language (ReLL)* (Alarcón and Wilde 2010) is another attempt to overcome this limitation of WADL by focusing on resources and links as the most important aspect of REST service description. Also WSDL 2.0 offers an explicit HTTP binding.

In summary, there is not yet an established machine-processable language for describing RESTful services, and there is not even consensus whether that would be useful or required, and if so, what should be described and what should be left open to avoid tight coupling. Most current RESTful APIs rely on HTML documentation.

In addition to how to describe the basic interaction with RESTful services (their resources, the representations, the uniform interface, the linking), another question that has been raised is how to describe services on a semantic level. The main goal there is to be able to find a service by searching on a semantic level, and the area of *Semantic Web Services* has received some attention, in particular in the areas of the *Semantic Web* and *Linked Data*. Many approaches are based on taking an existing service interface description language, and then augmenting this with semantic annotations, often in the form of RDF statements that are embedded into the service description. In such a scenario, it is possible to harvest RDF from a given set of service descriptions, and then apply standard Semantic Web methods to the resulting set of RDF data.

Generally speaking, the overlap between REST and Semantic Web activities is fairly small at the moment, though. This is caused by the fact that on a certain level of abstraction, the Semantic Web is simply a different set of constraints than those prescribed by REST. Instead of self-description, the representation metamodel is fixed and assumed to always be RDF, and REST's explicitly open-ended linking (links can point to resources identified by different URI schemes and thus implementing different uniform interfaces), the Semantic Web prescribes a more homogenous approach. Harmonizing the worlds of REST and the Semantic Web, or at least finding good ways to ensure mutually beneficial coexistence, is one of the current research challenges.

Composing RESTful Services

Service composition is one of the central tenets of service oriented computing, which – similar to service description – has been somewhat ignored in the context of the REST architectural style. The goal of service composition is to reuse existing services by means of assembling them in composite applications that combine them in novel and unexpected ways (Vinoski 2008b).

It is possible to apply composition to the REST architectural style in two ways. The first concerns the recursive construction of composite resources: resources which rely on other resources to manage parts of their state and delegate to other resources parts of their behavior. These can be implemented with languages such as the Business Process Execution Language (BPEL), which can be extended to support RESTful service composition as suggested in Pautasso (2009c). The second way makes use of the hypermedia constraint to push the execution of the actual composition back to the client. This is a significant departure from the encapsulation provided by traditional service composition as it relies on the client to pull together and compose data transitively linked from an initial composite representation.

On the Web, this is commonly achieved with so-called Mashup applications, which however provide a single user interface which gives an integrated view over multiple Web data sources and Web APIs (Daniel et al. 2007). The Mashup application itself is not usually delivered as a RESTful Web service so that it can be reused from other clients. Composite RESTful services instead are meant to be primarily reused as a service, since they do not necessarily aim at providing a user interface. Still, nothing prevents a composite RESTful service from using HTML as one of the representation formats for its composite resources, thus enabling the composite service to be accessed via a fully integrated, mashup-like user interface running in a Web browser.

Another important difference between Mashups and RESTful service composition concerns the degree in which the uniform interface of a RESTful Web service is used. Most mashup applications crawl multiple RESTful services to read, filter and aggregate their data but only very few ones are capable of pushing back updates to change the state of their component resources (Pautasso 2009a). This latter capability has important implications concerning the fault-tolerance of a composite RESTful Web service, as discussed in detail in Chap. 23.

Overall, whereas it is clear that REST needs to come to terms with service composition, there are still a number of interesting open issues that need to be worked out (Pautasso 2009b). On the one hand, existing service composition technology does not fit properly with RESTful Web services featuring a uniform interface without static formal descriptions, relying on dynamic content-type negotiation and on late binding to a highly dynamic set of resource identifiers. On the other hand, the recursive composition of uniform interfaces still needs to guarantee the validity of the safety and idempotency assumptions on which most of the “reliability” properties of REST are built upon.

About This Book

The original idea for this book came as a result of the *First International Workshop on RESTful Design (WS-REST 2010)* that the editors organized at the WWW2010 conference in May 2010 in Raleigh, NC. This workshop was the first attempt to raise awareness of REST as an important and relevant research topic for academia, but it was also well-attended by practitioners from the industry. The workshop generated an interesting set of papers looking at various perspectives of REST as a research topic (Pautasso et al. 2010), and this book is loosely based on some these workshop papers, but contains new contributions as well as extensively edited and extended versions of those papers which did appear in the proceedings.

This book is not intended to be an introduction to REST principles, or to be a practical guide on how to implement RESTful systems. There are already excellent books written about that. For example, *RESTful Web Services* (Richardson and Ruby 2007) by Leonard Richardson and Sam Ruby may be considered as the most influential book that started the movement of considering REST as a good choice for building loosely coupled and scalable services. Allamaraju (2010); Webber et al. (2010) should also not be missing from the shelf of readers interested in practical aspects of RESTful services development. The main goal of this book is to bridge the gap between the sometimes rather abstract work focusing only on research issues, and the sometimes not very disciplined approaches about how to design and implement something that qualifies as "being RESTful."

We would like to express our gratitude to all the authors and the reviewers of the chapters for their contribution to making this book a reality.

Outline of the Book

Foundations explores some of the foundations of the REST architectural style, both in terms of trying to frame the style in a way that allows to understand the main constraints and their effects on systems design, and in terms of contrasting the style with the other major style for designing and implementing Web services, the WS-* approach.

The part about *Design* has several chapters discussing the important issue of how to produce a system design that is RESTful, and how to do so in a way that produces a "good" design. From the point of view of service-orientation, identifying services as components of a *Service-Oriented Architecture (SOA)* is the starting point, and then the challenging question is how to design and implement a system that is a "good" implementation of such a SOA starting point. An important part of this part are chapters investigating how several of the core REST constraints influence the quality of the resulting design, and how important those constraints are for the eventual system design.

Development Frameworks address the question how a RESTful system design can be implemented in a way that is both cost-efficient, but still does allow the system to evolve in ways which are important. Frameworks can either focus on building back-end architectures or supporting the development of UIs in the rapidly evolving world of end-user platforms. In both cases, REST plays an important role in providing an architecture that should allow to build scalable, decentralized, and shareable services.

Application Case Studies looks at some scenarios where RESTful approaches have been used. REST claims to provide certain advantages over the WS-* style of Web services, but as always, it is important to look at the scenario and the design goals before any comparisons can be made. In this part, several case studies present such a comparison, allowing readers to explore how the REST approach has worked in concrete projects.

REST and Pervasive Computing is one of the areas where RESTful (i.e., Web-inspired) designs have become popular. One important reason for that is that in many scenarios of pervasive or ubiquitous computing, loosely coupled architectures and decentralized designs are essential for being scalable in a world of many sensors and other Web-enabled devices. The "Internet of Things" has gained quite a bit of traction as a buzzword, and this part of the book contains chapters describing forward-looking architectures that make the step forward to implementing a "Web of Things" that is built on RESTful principles.

While REST is being used as the general style underlying the Web and as a guiding principle in an increasing number of SOA projects, there still are open questions and research issues. *REST Research* addresses a number of those open issues, such as how to decide on resource granularity, how to model metadata, and how to handle transactions.

Foundations

1. *The Essence of REST Architectural Style (Jaime Navon and Federico Fernandez)* Roy Fielding introduced REST as an architecture style but the experience of the last few years has shown that there are different interpretations about its essence. The concepts of Restful application and Resource Oriented Architecture and their relationships are still source of some debate. In this chapter we start from Fielding's proposal to build a more detailed model of the REST architectural style and then we analyze the model through influence diagrams. The resulting model can be used to facilitate the understanding of this important architectural style and the effects and implications of relaxing one or more constraint. Finally we use the model to analyze and understand the main points of debate around ROA.
2. *REST and Web Services: In Theory and in Practice (Paul Adamczyk, Patrick H. Smith, Ralph E. Johnson, and Munawar Hafiz)*

There are two competing architectural styles employed for building Web services: RESTful services and services based on the WS-* standards (also known as “SOAP Web services”). These two styles have separate follower bases, but many differences between them are ideological rather than factual. In order to promote the healthy growth of Web services research and practice, it is important to distinguish arguments for implementation practices over abstract concepts represented by these styles, carefully evaluating the respective advantages of RESTful and WS-* Web services. Understanding these distinctions is especially critical for the development of enterprise systems, because in this domain, tool vendors have preferred WS-* services to the neglect of RESTful solutions. This chapter evaluates some of the key questions regarding the real and perceived distinctions between these two styles of Web services. It analyzes how the current tools for building RESTful Web services embody the principles of REST. Finally, it presents select open research questions to further the growth of RESTful Web services.

Design

3. *Designing a RESTful Domain Application Protocol (Ian Robinson)*
   This chapter discusses the significance of domain application protocols in distributed application design and development. Describing an application as an instance of the execution of a domain application protocol, it shows how we can design RESTful APIs that allow clients to drive the execution of a domain application protocol without binding to the protocol itself. The second half of the chapter provides a step-by-step example of a RESTful procurement application; this application realizes a procurement protocol in a way that requires clients to couple simply to media types and link relations, rather than to the protocol.
4. *Designing Hypermedia Engines (Mike Amundsen)*
   In this chapter, a number of different notions of hypermedia along with a formal definition of “Hypermedia Type” will be presented. In addition, nine Hypermedia Factors (H-Factors) that can be found in resource representations are identified and examples of these factors are provided. Armed with these nine H-Factors, several registered media types are analyzed to determine the presence of these hypermedia elements and to quantify the hypermedia support native to these media types. Finally, a prototypical media type (PHACTOR) is defined and reviewed in order to show how H-Factors can be incorporated into a media type in order to produce a data format that can act as an engine of application state.
5. *Beyond CRUD (Ivan Porres and Irum Rauf)*
   REST web services offer interfaces to create, retrieve, update and delete information from a database (also called CRUD interfaces). However, REST web services can also be used to create rich services that offer more than simple CRUD operations and still follow the REST architectural style. In such a case, it is important to creates and publish behavioral service interfaces that developers

can understand in order to use the service correctly. In this chapter, we explain how to use models to design rich REST services. We use UML class diagrams and protocol state machines to model the structural and behavioral features of rich services. The conceptual resource model that represents the structural feature adds addressability and connectivity features to the designed interface. The uniform interface feature is offered by constraining the invocation methods in the state machine to HTTP methods. In addition, to provide the feature of statelessness in our interface we use a state machine for behavioral modeling. This oxymoron is addressed by taking advantage of the fact that state invariants can be defined using query method on resources and the information contained in their response codes. The rich behavioral specifications present in the behavioral model show the order of method invocations and the conditions under which these methods can be invoked along with the expected conditions. We use this behavioral model to generate contracts in the form of preconditions and postconditions for methods of an interface.The design approach is implemented in Django web framework and the contracts generated from the behavioral model are asserted as contracts in the implemented interface. A proxy interface is also implemented in Django as a service monitor.

6. *Quantifying Integration Architectures (Jan Algermissen)*
   In a competitive environment, IT systems must be able to quickly respond to new business requirements. A sufficient level of simplicity and loose coupling can only be maintained by choosing the right integration styles. This chapter introduces a metric for quantifying integration architectures that can be used to guide strategic architectural decisions.
7. *FOREST: An Interacting Object Web (Duncan Cragg)*
   FOREST is a distributed and concurrent object architecture. In FOREST, objects set their state as a function of their current state plus the state of other objects observed through links. This observation occurs through either pull or push of linked object state. Such a programming model is declarative in nature, and thus very expressive, as well as being naturally concurrent. More importantly, it maps directly to RESTful distribution over HTTP, using GET for pull and POST for push of object state, in both directions between interacting servers. Objects are published into a global interacting object Web which can be described as "hyperdata." This mapping of object interaction into RESTful distribution leads to a symmetric re-interpretation of the hypermedia constraint to "hyperdata as the engine of hyperdata".

Development Frameworks

8. *Hypermedia-Driven Framework for Scalable and Adaptive Application Sharing (Vlad Stirbu and Juha Savolainen)*
   This chapter describes our experiences designing a solution for scalable and adaptive sharing of desktop and mobile applications, using a lightweight

network-based system compliant with the REST architectural style. The system delivers consistency of the rendered user interfaces with the state of the application logic using a stateless networking substrate. We describe the architecture focusing on how to model the user interfaces as a set of Web resources. Then, we present the prototype that implements the functionality as an extension of the Qt framework, which works with different Qt-based user interface toolkits. Finally, we present a multi-display and multi-user Texas Hold'em application that shows how the system is used in practice.

9. *RESTful Service Development for Resource-Constrained Environments (Amirhosein Taherkordi, Daniel Romero, Romain Rouvoy, and Frank Eliassen)*
   The use of resource-constrained devices, such as smartphones and Wireless Sensor Networks (WSNs) is spreading rapidly in our daily life. Accessing services from such devices is very common in ubiquitous environments, but mechanisms to implement and distribute these services remains a major challenge. Web services have been characterized as a widely-adopted approach to overcome heterogeneity, while this technology is still heavyweight for resource-constrained devices. The emergence of REST architectural style as a lightweight interaction model has encouraged researchers to study the feasibility of exploiting REST principles to integrate services hosted on devices with limited capabilities. In this chapter, we discuss the state-of-the-art in applying REST concepts to develop Web services for WSNs and smartphones, and then we provide a comprehensive survey of existing solutions in this area. In this context, we report on the DIGIHOME platform, a home monitoring middleware solution, which enables efficient service integration in ubiquitous environments using REST architectural style. In particular, we target our reference platforms for home monitoring systems, namely WSNs and smartphones, and report our experiments in applying the concept of Component-Based Software Engineering (CBSE) in order to provide resource-efficient RESTful distribution of Web services for those platforms.
10. *A REST Framework for Dynamic Client Environments (Erik Albert and Sudarshan Chawathe)*
   We describe methods for building RESTful applications that fully exploit the diverse and rich feature-sets of modern client environments while retaining functionality in the absence of these features. For instance, we describe how an application may use a modern JavaScript library to enhance interactivity and end-user experience while also maintaining usability when the library is unavailable to the client (perhaps due to incompatible software). These methods form a framework that we have developed as part of our work on a Web application for presenting large volumes of scientific datasets to non-specialists. The REST Framework for Dynamic Client Environments (RFDE) is a method for building RESTful Web applications that fully exploit the diverse and rich feature-sets of modern client environments while retaining functionality in the absence of these features. For instance, we describe how an application may use a modern JavaScript library to enhance interactivity and end-user experience while also maintaining usability when the library is unavailable to the client

(perhaps due to incompatible software). These methods form a framework that we have developed as part of our work on a Web application for presenting large volumes of scientific data to non-specialists.

11. *From Requirements to a RESTful Web Service: Engineering Content Oriented Web Services with REST (Petri Selonen)*
    This chapter presents an approach for proceeding from a set of requirements to an implemented RESTful Web service for content oriented systems. The requirements are captured into a simple domain model and then refined into a resource model. The resource model re-organizes the domain concepts into addressable entities: resources and interconnecting links, hypermedia representations, URIs and default HTTP operations and status codes. The approach has emerged from the experiences gained during developing RESTful Web services at Nokia Research Center.
12. *A Framework for Rapid Development of REST Web Services for Integrating Information Systems (Lars Hagge, Daniel Szepielak, and Przemyslaw Tumidajewicz)*
    Integrating information systems and legacy applications is a frequently occurring activity in enterprise environments. Service Oriented Architecture (SOA) and Web services are currently considered the best practice for addressing the integration issue. This chapter introduces a framework for rapid development of REST-based web services with a high degree of code reuse, which enables non-invasive, resource centric integration of information systems. It focuses on the general framework design principles and the role of REST, aiming to remain independent of particular implementation technologies. The chapter illustrates the framework's capabilities and describes experience gained in its application by examples from real-world information system integration cases.

Application Case Studies

13. *Managing Legacy Telco Data Using RESTful Web Services (Damaris Fuentes-Lorenzo, Luis Sánchez, Antonio Cuadra-Sanchez, and Mar Cutanda Rodríguez)*
    Our chapter aims to explain the activities to transform an existing collection of data into resources ready to be easily searched and queried, applying advanced web technologies such as RESTful web techniques. These technologies have been deployed in this work over traditional tools dealing with services offered to customers in a real Telecom company.
14. *Case Study on the Use of REST Architectural Principles for Scientific Analysis: CAMERA – Community Cyberinfrastructure for Advanced Microbial Ecology Research and Analysis (Abel Lin, Ilkay Altintas, Chris Churas, Madhusudan Gujral, Jeff Grethe, and Mark Ellisman)*
    The advent of Grid (and by extension Cloud) Computing along with Service Orientated Architecture (SOA) principles have led to a fundamental shift in the development of end-user application environments. No longer do stand-alone

applications need to be in- stalled on client workstations. Rather, user applications are now inherently lightweight – relying on remote service calls to "do the work". In the scientific domain, this loosely coupled; multi-tiered software architecture has been quickly adopted as raw data sizes have rapidly grown to a point where typical user workstations can no longer perform the necessary computational and data-intensive analyses. Here, we present the CAMERA (Community Cyberinfrastructure for Advanced Microbial Ecology Research and Analysis) project as a case study for a SOA in scientific research environments. Specifically, CAMERA is fundamentally based on a collection of REST services. These services are linked together by a scientific workflow environment (Kepler) and presented to end-users in a unified environment geared towards scientific genomic researchers.

15. *Practical REST in Data-Centric Business Applications: The Case of Cofidis Hispania (Jordi Fernandez and Javier Rodriguez)*
This chapter describes the migration of the IT environment in an important financial institution, from a mainframe-centric to a Web-centric environment in which the REST architectural style had a key role in the reference architecture that supported the new software development projects. The REST architectural style addressed the most critical constraints, contributing to address different software architecture challenges, both functional and non-functional.

REST and Pervasive Computing

16. *RESTifying Real-World Systems: A Practical Case Study in RFID (Dominique Guinard, Mathias Müller, and Vlad Trifa)*
As networked sensors become increasingly connected to the Internet, RFID or barcode-tagged objects are likely to follow the same trend. The EPC Network is a set of standards to build a global network for such electronically tagged goods and objects. Amongst these standards, the Electronic Product Code Information Service (EPCIS) specifies interfaces to capture and query RFID events from external applications. The query interface, implemented via SOAP-based Web services, enables business applications to consume and share data beyond companies borders and forms a global network of independent EPCIS instances. However, the interface limits the application space to the rather powerful platforms which understand WS-* Web services. In this chapter, we introduce tools and patterns for Web-enabling real-world information systems advertising WS-* interfaces. We describe our approach to seamlessly integrate RFID information systems into the Web by designing a RESTful (Representational State Transfer) architecture for the EPCIS. In our solution, each query, tagged object, location or RFID reader gets a unique URL that can be linked to, exchanged in emails, browsed for, bookmarked, etc. Additionally, this enables Web languages such as HTML and JavaScript to directly use RFID data to fast-prototype light-weight applications such as mobile applications or Web mashups. We illustrate these benefits by describing a JavaScript mashup platform that integrates with various

several services on the Web (e.g., Twitter, Wikipedia, etc.) with RFID data to allow managers along the supply chain and customers to get comprehensive data about their products.

17. *Leveraging the Web for a Distributed Location-Aware Infrastructure for the Real World (Vlad Trifa, Dominique Guinard, and Simon Mayer)*
    Since GPS receivers have become a commodity anyone could access and use location information simply and freely. Such an easy access to ones' location is instrumental to development of location-aware applications. However, existing applications are static in that they do not model relations between places and mobile things. Moreover, these applications do not allow to easily map the physical location of mobile devices to virtual resources on the Internet. We attempt to bridge this gap by extending the base concepts that make up the Internet with the physical location of devices, in order to facilitate the development of Web-based location-aware applications for embedded mobile devices. In this chapter, we propose a simple infrastructure for the "Web of Things" that extends the existing Web to enable location-aware applications. The proposed solution enables a natural hierarchical way to search for location-aware devices and the services they provide.
18. *RESTful Service Architectures for Pervasive Networking Environments (Mauro Caporuscio, Marco Funaro, and Carlo Ghezzi)*
    Computing facilities are an essential part of the fabric of our society, and an ever-increasing number of computing devices is deployed within the environment in which we live. The vision of pervasive computing is becoming real. To exploit the opportunities offered by pervasiveness, we need to revisit the classic software development methods to meet new requirements: (1) pervasive applications should be able to dynamically configure themselves, also benefiting from third-party functionalities discovered at run time and (2) pervasive applications should be aware of, and resilient to, environmental changes. In this chapter, we focus on the software architecture, with the goal of facilitating both the development and the run-time adaptation of pervasive applications. More specifically we investigate the adoption of the REST architectural style to deal with pervasive environment issues. Indeed, we believe that, although REST has been introduced by observing and analyzing the structure of the Internet, its field of applicability is not restricted to it. The chapter also illustrates a proof-of-concept example, and then discusses the advantages of choosing REST over other styles in pervasive environments.

REST Research

19. *On Entities in the Web of Data (Michael Hausenblas)*
    The chapter explores what "entities" in the Web of Data are. As a point of departure, we examine a number of widely used RESTful Web APIs in terms of URI-space-design and hyperlinking support in the offered resource

representations. Based on the insights gained from the API review, we motivate the concept of an entity as well as its boundaries. Eventually, we discuss the relevance of the entity concept for publishers and consumers of Web data, as well as the impact on Web data design issues.

20. *A Resource Oriented Multimedia Description Framework (Hildeberto Mendonca, Vincent Nicolas, Olga Vybornova, and Benoit Macq)*
    This chapter presents a multimedia archiving framework to describe the content of multimedia resources. This kind of content is very rich in terms of meanings and archiving systems have to be improved to consider such richness. This framework simplifies the multimedia management in existing applications, making it accessible for non-specialized developers. This framework is fully implemented on the REST architectural style, precisely mapping the notion of resource with media artifacts, and scaling to address the growing demand for media. It offers an extensive support for segmentation and annotation to attach semantics to content, helping search mechanisms to precisely index those content. A detailed example of the framework adoption by a medical imaging application for breast cancer diagnosis is presented.
21. *Metadata Architecture in RESTful Design (Antonio Garrote and María N. Moreno García)*
    This chapter is an overview of the role that metadata plays in the design of RESTful services and APIs. The chapter describes how metadata can be associated to resources using the HTTP protocol and other standard technologies like RDFa. Techniques for metadata extraction and metadata discovery are also introduced. The ultimate goal of the chapter is to provide tools to build truly self-describing RESTful resources.
22. *RESTful Services with Lightweight Machine-Readable Descriptions and Semantic Annotations (Jacek Kopecky, Tomas Vitvar, Carlos Pedrinaci, and Maria Maleshkova)*
    REST was originally developed as the architectural foundation for the human-oriented Web, but it has turned out to be a useful architectural style for machine-to-machine distributed systems as well. The most prominent wave of machine-oriented RESTful systems are Web APIs (also known as RESTful services), provided by Web sites such as Facebook, Flickr, and Amazon to facilitate access to the services from programmatic clients, including other Web sites. Currently, Web APIs do not commonly provide machine-processable service descriptions which would help tool support and even some degree of automation on the client side. This chapter presents current research on lightweight service description for Web APIs, building on the HTML documentation that accompanies the APIs. HTML documentation can be annotated with a microformat that captures a minimal machine-oriented service model, or with RDFa using the RDF representation of the same service model. Machine-oriented descriptions (now embedded in the HTML documentation of Web APIs) can also capture the semantics of Web APIs and thus support further automation for clients. The chapter includes a discussion of various types and

degrees of tool support and automation possible using the lightweight service descriptions.

23. *Towards Distributed Atomic Transactions Over RESTful Services (Guy Pardon and Cesare Pautasso)*
    There is considerable debate in the REST community whether or not transaction support is needed and possible. This chapter's contribution to this debate is threefold: we define a business case for transactions in REST based on the Try-Cancel/Confirm (TCC) pattern; we outline a very light-weight protocol that guarantees atomicity and recovery over distributed REST resources; and we discuss the inherent theoretical limitations of our approach. Our TCC for REST approach minimizes the assumptions made on the individual services that can be part of a transaction and does not require any extension to the HTTP protocol. A very simple but realistic example helps to illustrate the applicability of the approach.

References

Rosa Alarcón and Erik Wilde. RESTler: Crawling RESTful Services. In Michael Rappa, Paul Jones, Juliana Freire, and Soumen Chakrabarti, editors, *19th International World Wide Web Conference*, pages 1051–1052, Raleigh, North Carolina, April 2010. ACM Press, New York.

Subbu Allamaraju. *RESTful Web Services Cookbook*. O'Reilly & Associates, Sebastopol, California, February 2010.

Don Box, Gopal Kavivaya, Andrew Layman, Satish Thatte, and Dave Winer. SOAP: Simple Object Access Protocol. Internet Draft draft-box-http-soap-01, November 1999.

Tim Bray, Jean Paoli, and C. Michael Sperberg-McQueen. Extensible Markup Language (XML) 1.0. World Wide Web Consortium, Recommendation REC-xml-19980210, February 1998.

Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. World Wide Web Consortium, Recommendation REC-wsdl20-20070626, June 2007.

Luc Clement, Andrew Hately, Claus von Riegen, and Tony Rogers. UDDI Version 3.0.2. Organization for the Advancement of Structured Information Standards, UDDI Spec Technical Committee Draft, October 2004.

Florian Daniel, Maristella Matera, Jin Yu, Boualem Benatallah, Regis Saint-Paul, and Fabio Casati. Understanding UI Integration: A Survey of Problems, Technologies, and Opportunities. *IEEE Internet Computing*, 11(3): 59–66, May–June 2007.

Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, Irvine, California, 2000.

Marc Hadley. Web Application Description Language (WADL). Technical Report TR-2006-153, Sun Microsystems, April 2006.

Maria Maleshkova, Carlos Pedrinaci, and John Domingue. Investigating Web APIs on the World Wide Web. In *Proceedings of the 8th IEEE European Conference on Web Services (ECOWS2010)*, pages 107–114, December 2010.

Cesare Pautasso. Composing RESTful Services with JOpera. In Alexandre Bergel and Johan Fabry, editors, *International Conference on Software Composition 2009*, volume 5634 of *Lecture Notes in Computer Science*, pages 142–159, Zürich, Switzerland, July 2009. Springer-Verlag, Berlin, Heidelberg, New York.

Cesare Pautasso. On Composing RESTful Services. In Frank Leymann, Tony Shan, Willen-Jan van den Heuvel, and Olaf Zimmermann, editors, *Software Service Engineering*, number 09021 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, June 2009.

Cesare Pautasso. RESTful Web Service Composition with BPEL for REST. *Data & Knowledge Engineering*, 68(9): 851–866, September 2009.

Cesare Pautasso and Erik Wilde. Why is the Web Loosely Coupled? A Multi-Faceted Metric for Service Design. In Juan Quemada, Gonzalo León, Yoëlle S. Maarek, and Wolfgang Nejdl, editors, *18th International World Wide Web Conference*, pages 911–920, Madrid, Spain, April 2009. ACM Press, New York.

Cesare Pautasso, Erik Wilde, and Alexandros Marinos, editors. *First International Workshop on RESTful Design (WS-REST 2010)*, Raleigh, North Carolina, April 2010.

Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. RESTful Web Services vs. "Big" Web Services: Making the Right Architectural Decision. In Jinpeng Huai, Robin Chen, Hsiao-Wuen Hon, Yunhao Liu, Wei-Ying Ma, Andrew Tomkins, and Xiaodong Zhang, editors, *17th International World Wide Web Conference*, pages 805–814, Beijing, China, April 2008. ACM Press, New York.

Paul Prescod. Roots of the REST/SOAP Debate. In *2002 Extreme Markup Languages Conference*, Montréal, Canada, August 2002.

Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly & Associates, Sebastopol, California, May 2007.

Steve Vinoski. RPC and REST: Dilemma, Disruption, and Displacement. *IEEE Internet Computing*, 12(5): 92–95, September 2008.

Steve Vinoski. Serendipitous Reuse. *IEEE Internet Computing*, 12(1): 84–87, January 2008.

Jim Webber, Savas Parastatidis, and Ian Robinson. *REST in Practice: Hypermedia and Systems Architecture*. O'Reilly & Associates, Sebastopol, California, September 2010.

Part I
Foundations

Chapter 1
The Essence of REST Architectural Style

Jaime Navon and Federico Fernandez

Abstract There is an increasing interest in understanding and using REST architectural style. Many books and tools have been created but there is still a general lack of understanding its fundamentals as an architecture style. The reason perhaps could be found in the fact that REST was presented in a doctoral dissertation, with relatively high entry barriers for its understanding, or because the description used models that were more oriented towards documentation than to working practitioners.

In this chapter we examine, in a systematic manner, some of the issues about Fielding's doctoral dissertation that have caused so much confusion. We start examining REST as an architecture style as a sequence of architectural decisions. We use then influence diagrams to build a model that allows us to see how the architectural decisions take us from classic architectural styles like client-server and layered-system to REST. The graphical model not only facilitates the understanding of this important new architectural style, but also serves as a framework to assess the impact of relaxing or adding more constraints to it. As a final example we analyze the resource-oriented architecture (ROA) to find out one important constraint that is present in REST is missing in ROA and this has an impact on both scalability and modifiability.

Introduction

REST is usually referred to, as it was originally introduced in the Chap. 5 of the Ph.D. dissertation of Dr. Roy Fielding: an architectural style (Fielding 2000). To fully understand the idea it is necessary to read the full dissertation since REST

J. Navon (✉)
Department of Computer Science, Universidad Catolica de Chile, Santiago, Chile
e-mail: jnavon@ing.puc.cl

rationale cannot be understood without the definitions and concepts of Chaps. 1–3 and the description of the WWW architecture requirements of Chap. 4. The problem is, that even after reading the complete dissertation you might still have questions related more to the real-world implications than to abstract theoretical software engineering issues.

Why is that, in spite of the huge success of REST, there is still so much debate about whether a given service API should be considered REST? What is the difference between REST and Restful? What is the relationship between REST and resource oriented architecture (ROA)?

First there is this slippery thing called Architectural Style. Fielding defines it as a "coordinated set of architectural constraints that restricts the roles/features of architectural elements and the allowed relationships among those elements within any architecture that conform to that style". This is why, consistent with this definition, he introduces REST precisely trough a set of constraints, namely client–server, stateless, cache, uniform interface, etc.

Explaining REST by introducing constraints associated to a number of primitive architectural styles is just fine for a doctoral dissertation but it leaves a lot of room to interpretation. This is the source of some heated debates about concrete architectures that might be "betraying" the REST principles.

Richardson and Ruby (2007) in their book present what they call a "*ROA*" as a simple set of guidelines that guarantees a RESTful architecture. They make clear though that there are other concrete architectures that may also be RESTful.

Not only is the concept of architectural style is problematic, there is no complete agreement on what is really software architecture. As defined by the Institute of Electrical and Electronics Engineers (IEEE) Recommended Practice for Architecture Description of Software-Intensive Systems (IEEE standard 1471–2000), architecture is "*the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution*". This definition is fairly abstract and applies to systems other than just software. Meanwhile, Fielding emphasizes that software architecture is an abstraction of the run-time behavior of a software system and not just a property of the static software source code.

Bass et al. (2003) define software architecture as "structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them". The plural in structures acknowledges the possibility of more than one structure, each of them conveying architectural information. Some may be more related to the static structure and some more to the dynamic aspects.

We believe that Roy Fielding's dissertation is indeed the key document to dive deeper into the essence of REST; a document that, as Martin Fowler puts forward in his foreword in a recent book (Webber et al. 2010), "is far more often referred to than it is read". We hope that this chapter will contribute to a better understanding of REST and also as a framework to evaluate and discuss new proposals of software architectures and architecture styles.

Architectural Styles and Architectural Properties

As we said before, Fielding definition of architectural style involves architectural restrictions. Furthermore, he suggests that the space of all possible architectural styles can be seen as a derivation tree, where nodes are derived from others by adding new restrictions. Some of these nodes correspond to well known or "basic" architectural styles, whereas others will be hybrid nodes corresponding to combinations or derivations from the basic styles. Traversing the tree from the root to a node would allow us to understand all the design decisions associated to a specific style be it basic or derived. A concrete architecture will adhere more or less to one of these architectural styles depending on how close it is to the cumulative design decisions associated to the corresponding node in the derivation tree. Since each style induces a set of architectural properties, traversing the tree provides us with a good understanding of the architectural properties that our concrete or specific architecture will exhibit once it is implemented.

Fielding uses a qualitative approach to compare some of the most important architectural styles. For each of these styles, identified by a short symbolic name ("Pipe and Filter" is PF, "Client–Server" is CS, etc.), a plus or a minus sign is assigned depending on whether the style under consideration has a positive or negative impact on the software quality, that is on the architecture properties. Sometimes a style may affect a software quality both positively and negatively (because software qualities listed are relatively coarse grained, for the sake of simplicity and visualization).

The impacts of each architectonic style on each quality is presented by Fielding as a table in which there is also information about what styles represent derivations from other styles. Unfortunately, not all the styles or constraints that are part of the derivation of a style are shown in the table. Therefore, the impact of a derived style on the set of software qualities (the plus and minus signs in each row) is not always a simple union of the impact of its predecessors. Figure 1.1 shows the complete derivation for the REST architectural style. This derivation, the impact table, and some additional explanations are used in the dissertation to describe REST rationale. Table 1.1 is a slightly modified version of the original table. We filled some incomplete cells and added two new rows: one for Uniform Interface (U) and one for REST. The marked cells are those whose signs do not correspond to any possible union of the styles they derive from.

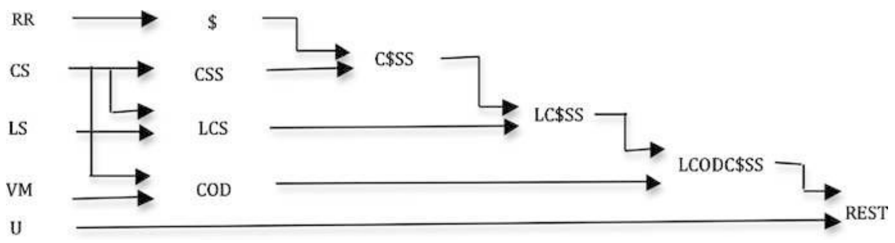


Fig. 1.1 REST derivation tree

Table 1.1 Impact table, extended

Style	Derivation	Net performance	UP perform.	Efficiency	Scalability	Simplicity	Evolyability	Extensibility	Customization	Configuration	Reusability	Visibility	Portability	Reliability
PF			+−			+	+	+		+	+			
UPF	PF	−	+−			++	+	+		++	++	+		
RR			++		+									+
$	RR		++	+	+	+								🟥
CS					+	+	+							
LS			−		+		+							
LCS	CS + LS		−		++	+	++				+		+	
CSS	CS	−			++	+	+					+		+
C$SS	CSS + $	−	+	+	++	+	+					+		+
LC$SS	LCS+C$SS	−	+−	+	+++	++	++				+	+	+	+
RS	CS			+	−	+	+					−		
RDA	CS			+	−	−	🟥					+		−
VM						+−		+				−	+	
REV	CS + VM			+	−	+−	🟥	+	+			−	+	−
COD	CS + VM		+	+	+	+−		+	+	+		−		
LCODC$S	LC$SS+CO	−	++	++	+4+	++−+	++	+	+	+	+	+−	+	+
MA	REV + COD		+	++		+−		++	+	+		−	+	🟥
EBI				+	–	+−	+	+		+	+	−		−
C2	EBI + LCS		−	+		+	++	+		+	++	+−	+	+−
DO	CS + CS	−		+	🟥	🟥	+	+		+	+	−		−
BDO	DO + LCS	−	−	🟥	🟥	🟥	++	+		+	++	−	+	
U				−		+	+					+		
REST	LC0DC$SS	−	++	++−	+4+	++−++	+++	+	+	+	+	+−+	+	+

The REST entry in the table reflects the fact that this architectural style can be derived from several styles (see Fig. 1.1).

Towards a Model for REST

Fielding describes REST by defining the architectural elements (instances of components, connectors and data) present in REST, and using a process view to show some possible configurations of these elements. This model is useful to describe an architectural style, but it is not practical for understanding its design rationale. What is needed is some sort of representation of REST as a set of constraints.

Inspired by the concept of derived styles we explained before, we will describe REST by using the concept of *architectural decision*. An architectural decision is a named set of constraints that can be added to an architectural style. The result of adding an architectural decision to an architectural style is another architectural style. A corollary of this definition is that a given architectural decision can only be made over certain architectural styles. For example, we could say that the architectural decision called "Stateless CS Interactions" only makes sense if applied over the Client–Server architectural style. What we gain with the introduction of this concept is that we can now describe an architectural style as a sequence of architectural decisions. This is somehow similar to the derived style approach used by Fielding but making every component of an architectural style explicit. Since the properties of an architectural style become the cumulative properties of its individual architectural decisions, it can be helpful for examining REST design rationale.

Going one step further, what we really want is a model of the REST design rationale that we could manipulate to visualize changes in a concise manner. Here are the ideal requirements for such a model:

- (R1) Visualize and understand how each one of the architectural decisions of REST impacts the set of goals that guided its design.
- (R2) Visualize and understand the changes caused in the induced properties if new architectural decisions are added to REST, or if existing decisions in REST are replaced for others.
- (R3) Visualize the set of alternative architectural decisions for each decision in REST.
- (R4) Easily modify the REST model to visualize and understand how each one of the architectural decisions of REST would impact a different set of goals.
- (R5) The model should be a loyal representation of the dissertation idea of REST.

There were several possible options. We could extend the classification framework of Table 1.1 by adding rows at the top grouping the different properties into broader categories, as a hierarchy of desired properties. This approach would satisfy the visualization part of R1 and R2, and maybe R4, but would not improve much in terms of understandability. We could instead represent REST as a set of documents, one per architectural decision, containing a description of the decision and the alternatives discarded (Jansen and Bosch 2005) but it wouldn't satisfy requirements R1, R2, and R4.

A better choice is to use a simplified version of *influence diagrams*, a graphical language defined in Johnson et al. (2007). We only need three types of node (*utility*, *chance,* and *decision)* and one type of arrow that would mean different things depending on the type of the connected nodes. As we are not trying to make a model capable of probabilistic reasoning, the resulting graphical notation is very close to the one explained in Chung et al. (1999).

This graphical language allows us to satisfy most of the requirements. The visibility part of R1 and R2 is met using collapsible graphical elements. The understandability part would is met by adding as many kinds of boxes as necessary to trace the impact of a decision over the set of goals. By drawing hierarchies of goals, from the most general ones to those represented by software qualities we can satisfy R4. Finally, extracting only from the dissertation all the knowledge used to define the elements of the diagram and their relationships, and documenting all these definitions with comments in the diagram we satisfy R5. The only requirement that would not be satisfied is R3, but the ability to trace causality from decisions to goals should help the user identify possible alternatives for each decision.

Analysis of REST Trough Influence Diagrams

Although not necessary, it is nice to use a software tool to build the diagrams. We used *Flying Logic*[1] flexible visual modeling tool. This is a commercial product but there is a free reader available so people who only want to read de diagrams do not need to buy the full product.

The influence diagram nodes corresponding to utility, decisions and chances were represented by tagged boxes (Fig. 1.2). The little circles present in some arrows and boxes denote commentaries (we used them to copy fragments from the dissertation). A black arrow, between a decision and a chance, means that the decision has a positive causal relationship with the chance whereas a grey arrow means that the decision has a negative causal relationship with the chance.

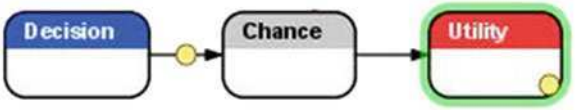


Fig. 1.2 Notation of the influence diagrams

[1] http://www.flyinglogic.com

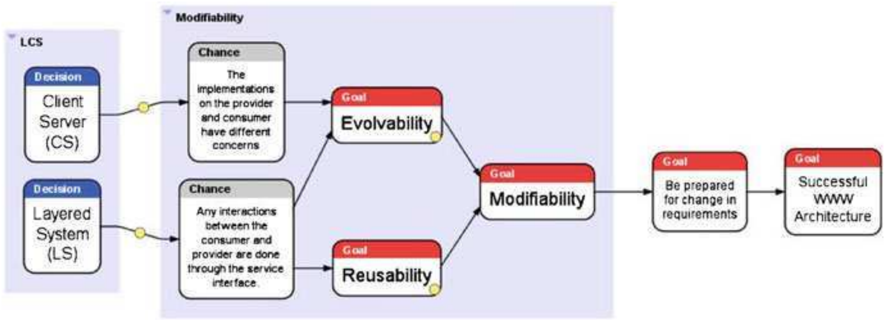

LCS
Modifiability
Decision
Client Server (CS)
Decision
Layered System (LS)
Chance
The implementations on the provider and consumer have different concerns
Chance
Any interactions between the consumer and provider are done through the service interface
Goal
Evolvability
Goal
Reusability
Goal
Modifiability
Goal
Be prepared for change in requirements
Goal
Successful WWW Architecture

Fig. 1.3 Sample of REST influence diagram

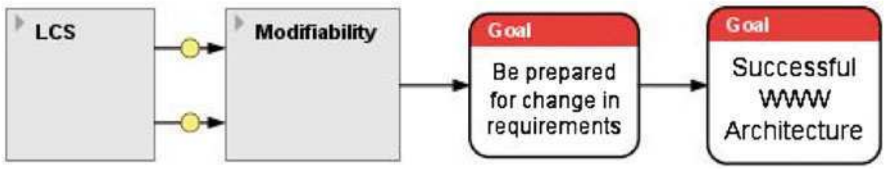

LCS
Modifiability
Goal
Be prepared for change in requirements
Goal
Successful WWW Architecture

Fig. 1.4 Sample of REST influence diagram, collapsed

An arrow between a chance and a utility can only be black and denotes a positive causal relationship between the chance and the utility node. An arrow between two utilities means that the first utility is a child of the second one. Arrows between decisions were used to denote temporal precedence.

Because of limitations in the tool, we do not use arrows to connect decisions. Instead, we describe the sequence of decisions that defines REST as a hierarchy of decision sets. This improves visibility at the cost of reduced understandability. If a decision is outside a given set, it means that it was made after the decisions inside the set, but if two or more decisions are in the same set, the user could not know which was done before the other, so he would have to get this information from another source (e.g. comments).

Figure 1.3 shows part of the REST influence diagram. Each decision represents an architectural style used in the derivation of REST, and each chance explains why a given decision affects a given utility. The dashed rectangles in the diagram can collapse its contents while maintaining the arrows going from the group to the outside (Fig. 1.4 shows the result of collapsing both named rectangles).

A more complete influence diagram for REST in the condensed mode is presented in Fig. 1.5. If we delete all the chances and decisions from the diagram and expand all the utility nodes, we get a horizontal version of the tree (Fig. 1.6).

To produce the chances we performed an exhaustive revision of the dissertation. Once we identified why a decision was related to a utility node, either we created a new chance between both nodes or we reused an already existent one. In order to maximize the reuse of chances, we tried to generalize each chance as much as possible.

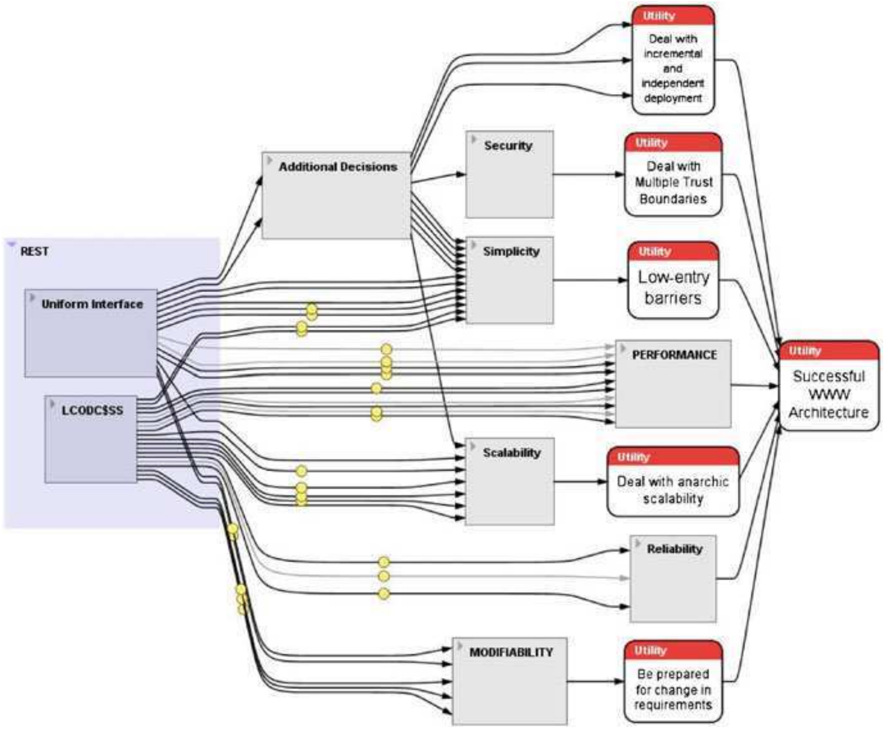

REST
Uniform Interface
LCODC$SS
Additional Decisions
Security
Simplicity
PERFORMANCE
Scalability
Reliability
MODIFIABILITY
Utility
Deal with incremental and independent deployment
Deal with Multiple Trust Boundaries
Low-entry barriers
Deal with anarchic scalability
Be prepared for change in requirements
Successful WWW Architecture

Fig. 1.5 Condensed view of REST influence diagram

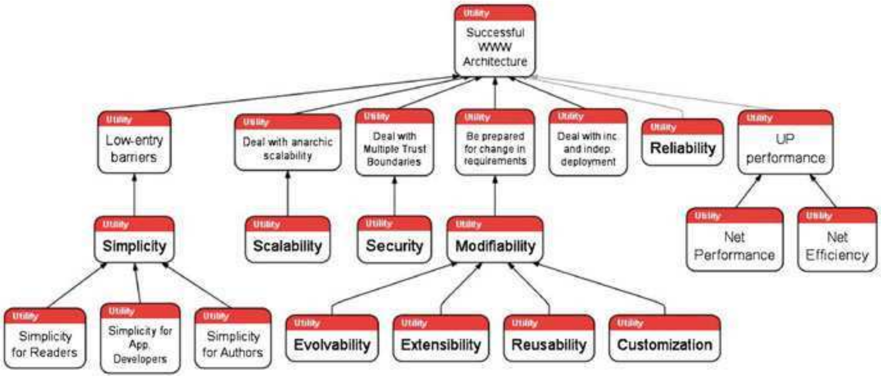


Fig. 1.6 Goals of the standard WWW architecture

Relating general network based software qualities with the goals of the WWW architecture is not straightforward. Furthermore, to make the diagram easier to read and understand, we eliminated some redundant utility nodes (portability), transformed others into chances (visibility), divided some utilities into lower

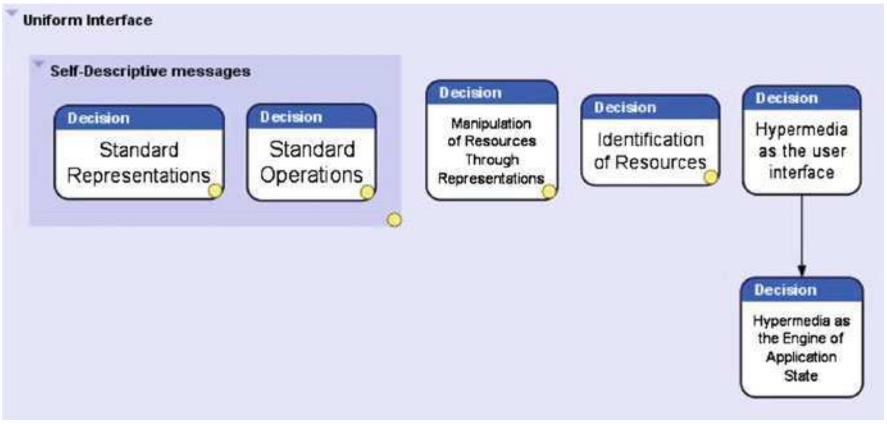

Uniform Interface
Self-Descriptive messages
Decision
Standard Representations
Decision
Standard Operations
Decision
Manipulation of Resources Through Representations
Decision
Identification of Resources
Decision
Hypermedia as the user interface
Decision
Hypermedia as the Engine of Application State

Fig. 1.7 Architectural decisions of the Uniform Interface

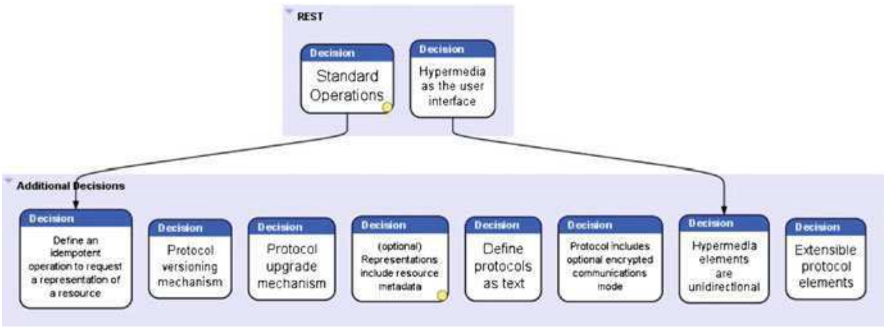

REST
Decision
Standard Operations
Decision
Hypermedia as the user interface
Additional Decisions
Decision
Define an idempotent operation to request a representation of a resource
Decision
Protocol versioning mechanism
Decision
Protocol upgrade mechanism
Decision
(optional) Representations include resource metadata
Decision
Define protocols as text
Decision
Protocol includes optional encrypted communications mode
Decision
Hypermedia elements are unidirectional
Decision
Extensible protocol elements

Fig. 1.8 Architectural decisions outside REST

level utility nodes (simplicity) and rearranged the hierarchy of utility nodes (UP Performance, Net Performance and Net Efficiency).

Initially, every decision node corresponded to one of the styles used in the derivation of REST, but then we decided to include decisions of lower granularity by dividing some styles into smaller parts.

Figure 1.7 shows the decisions that compose the Uniform Interface in the REST influence diagram. The decisions we included are Standard Representations, Standard Operations, and Hypermedia as the User Interface.

While developing the diagram, we also extracted some decisions and corresponding chances that were not explicitly explained as parts of REST derivation. Those decisions are shown in Fig. 1.8. Some of them can be considered constraints that can be added to REST to characterize a more restricted architectural style.

For example, the decision to define an idempotent operation X to request a representation of a resource could be seen as a constraint to be added to REST defining a new architectural style. Any software architecture conforming to X would also be an instance of REST.

The influence diagram then, can facilitate the task of checking what happens if we add constraints to REST. All that is needed is to write the constraint as a decision node, find the chances and utilities that it impacts, and reason about its usefulness.

The coding of the REST architectural style into the diagrams we presented before allows us to reason in a much more easy and precise way. Here are just a few observations:

- Simplicity and scalability are the two goals that receive more positive effects from REST. Modifiability and performance follow although the last one is also affected negatively by some decisions.
- Security is not addressed directly by REST. It is only treated in a lower abstraction level, and only by one chance: "Protocol includes optional encrypted communications mode" which in turn affects the sub-utility node: "network-level privacy".
- Although the property of Reliability has four chances, REST affects positively to only two of them, both coming from the Client–Stateless–Server style.
- It is clear that Fielding was thinking in human users rather than bots. This is manifested, for example, in the chances affecting the goal of Simplicity that had to be divided into: "Simplicity for Authors," "Simplicity for Readers," and "Simplicity for Application Developers". One practical consequence of this is that if we want to make an assessment of how REST induces simplicity in SOA, we should start by classifying the users of SOA and redistributing the chances related to the sub-utilities of Simplicity in the REST diagram, into the sub-utilities defined for SOA.

ROA Under the Magnifying Glass

The ROA as defined in Richardson and Ruby (2007) is "a way of turning a problem into a RESTful web service: an arrangement of URIs, HTTP, and XML that works like the rest of the Web, and that programmers will enjoy using." Another term used for the same purpose is Web Oriented Architecture, which is used by some IT consultants to name the application of the WWW standard protocols and proven solutions for the construction of distributed software systems (Hinchcliffe 2008).

Recently, it became clear that some of these approaches were not following all of the REST constraints, so practitioners started to debate, not only about ways to follow those constraints, but also about how important was to follow all of them.

To further test the applicability of our REST model, we will try now to provide an answer to a recent question that has been subject of a strong: *Considering the*

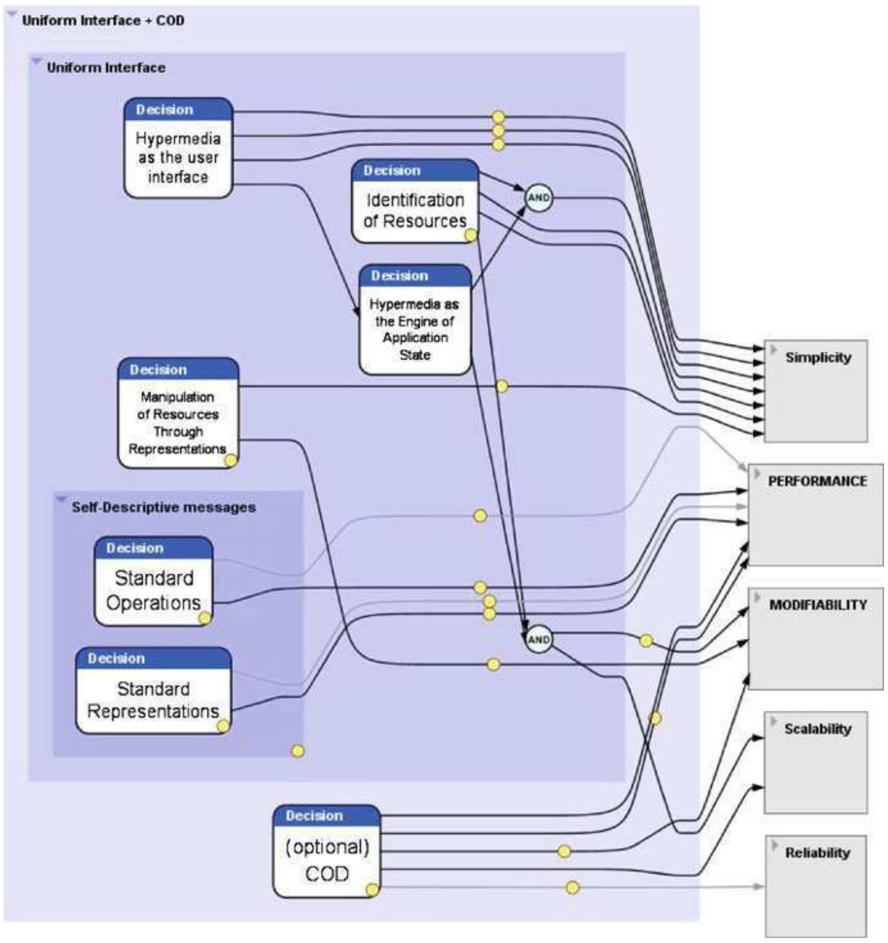


Fig. 1.9 Properties of COD and the uniform interface

recent clarifications[2,3,4] *regarding misunderstandings among practitioners of ROA, what properties are missing in ROA?*

The Influence Diagram of REST is a useful tool to identify what properties are lost by not following all the constraints. To accomplish this task, we take the REST influence diagram and delete all decision nodes that are not relevant. These decisions are those represented by the LC$SS architectural style and by those decisions used for the development of HTTP and URI that do not correspond to the core of REST. The resulting diagram is shown in Fig. 1.9.

[2]http://roy.gbiv.com/untangled/2008/on-software-architecture
[3]http://roy.gbiv.com/untangled/2009/it-is-okay-to-use-post
[4]http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

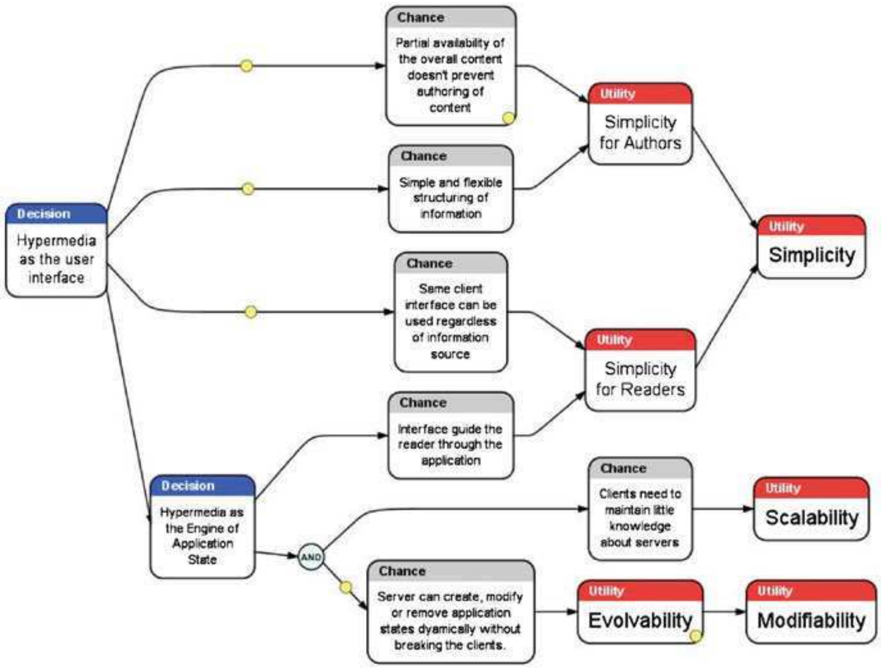

Chance
Partial availability of the overall content doesn't prevent authoring of content
Chance
Simple and flexible structuring of information
Decision
Hypermedia as the user interface
Chance
Same client interface can be used regardless of information source
Chance
Interface guide the reader through the application
Decision
Hypermedia as the Engine of Application State
AND
Chance
Clients need to maintain little knowledge about servers
Chance
Server can create, modify or remove application states dyamically without breaking the clients.
Utility
Simplicity for Authors
Utility
Simplicity for Readers
Utility
Simplicity
Utility
Scalability
Utility
Evolvability
Utility
Modifiability

Fig. 1.10 Properties of Hypermedia

ROA lacks only one decision: "Hypermedia as the engine of application state". There is also an optional decision: "Hypermedia as the user interface" (resource representations should be connected by links).

If we focus only in the arrows going from these two decisions we get the diagram of Fig. 1.10. Clearly, the constraint "Hypermedia as the user interface" was added to REST as a means to lower the WWW entry barriers for human readers and authors. This explains why it didn't make much sense for developers thinking rather in machine interaction.

On the other hand, "Hypermedia as the engine of application state" affects two chances that would be kept in the diagram even if we think that the user is a machine, and these chances impact Scalability and Evolvability.

So, why did the ROA proponents are ready to relax this constraint? Because in the context of machine-to-machine integration, it may require representations containing semantic hypermedia and this in turn would negatively affect the utilities "Simplicity for Application Developers" and "Performance". The additional effort is not considered worthwhile when compared to the benefits of Scalability and Modifiability.

The answer to the original question, thus, is that by not following the hypermedia constraint, the architectures conforming to ROA are indeed less scalable and modifiable than architectures conforming to REST.

We do not know exactly how less scalable and modifiable would be. The tradeoff of adding these constraints to ROA at the cost of lower simplicity and performance must be the subject of further research. In fact, one interesting research line could explore a way to modify the influence diagram technique to provide not only qualitative but also quantitative assessments, to help the architect to reason more effectively about the impact of different architectural styles on the desired set of software qualities.

The result obtained through REST influence diagram may not be surprising to REST practitioners, but now we can guarantee that it is founded in the knowledge included in the dissertation and nothing else. This is one of the most valuable properties of this model. A second valuable property is that adding decisions and chances can easily extend the model. For example, one could change both decisions in Fig. 1.10 by one group called “Semantic hypermedia as the engine of application state”, and add to that group decisions like “Machine-readable hypermedia as the client interface” and “Shared semantic data model between Client and Server,” thus starting to build a new architectural style that reuses part of REST structure and design rationale.

References

Bass, L., Clements, P., and Kazman, R. (2003) Software Architecture in Practice, 2nd Ed., Addison Wesley Professional, Reading, MA, USA.

Chung, L., Nixon, B. A., Yu, E., and Mylopoulos, J. (1999) Non-Functional Requirements in Software Engineering. International Series in Software Engineering, Vol. 5, Springer, Berlin, Heidelberg, New York.

Fielding, R. T. (2000) *Architectural styles and the design of network-based software architectures*. Ph.D. Dissertation, University of California, Irvine.

Hinchcliffe, D. (2008) What is WOA? It’s the Future of Service-Oriented Architecture (SOA). *Dion Hinchcliffe’s Blog – Musings and Ruminations on Building Great Systems*. Retrieved January 11th, 2008, http://hinchcliffe.org/archive/2008/02/27/16617.aspx.

Jansen, A. and Bosch, J. (2005) Software Architecture as a Set of Architectural Design Decisions. Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture, WICSA ’05, Pittsburgh, PA, USA.

Johnson, P., Lagerström, R., Närman, P., and Simonsson, M. (2007) Enterprise architecture analysis with extended influence diagrams. *Information Systems Frontiers, 9* (2–3). doi: 10.1007/s10796–007–9030-y.

Richardson, L. and Ruby, S. (2007) Restful web services. O’Reilly Media Inc. USA.

Webber, J., Parastaditis, S., and Robinson, I. (2010) Rest in Practice, O”Reilly Media Inc., USA.

Chapter 2
REST and Web Services: In Theory and in Practice

Paul Adamczyk, Patrick H. Smith, Ralph E. Johnson, and Munawar Hafiz

Abstract There are two competing architectural styles employed for building Web services: RESTful services and services based on the WS–∗ standards (also known as "SOAP Web services"). These two styles have separate follower bases, but many differences between them are ideological rather than factual. In order to promote the healthy growth of Web services research and practice, it is important to distinguish arguments for implementation practices over abstract concepts represented by these styles, carefully evaluating the respective advantages of RESTful and WS–∗ Web services. Understanding these distinctions is especially critical for the development of enterprise systems, because in this domain, tool vendors have preferred WS–∗ services to the neglect of RESTful solutions. This chapter evaluates some of the key questions regarding the real and perceived distinctions between these two styles of Web services. It analyzes how the current tools for building RESTful Web services embody the principles of REST. Finally, it presents select open research questions to further the growth of RESTful Web services.

Introduction

Since its inception, the Web has been an open frontier of exploration in software and network system design. New ideas were tried and tested first, but organized and standardized later, once they proved their utility. For example, HTTP, the transport protocol of the Web, had been in use for more than half a decade before its state of practice was written down as HTTP/1.0 (Berners-Lee et al. 1996) in May 1996. But the standardization process continued until 1999, when the final revision of HTTP/1.1 (Fielding et al. 1999) standard was completed. The architectural principles behind HTTP and other Web standards were described by

P. Adamczyk (✉)
Booz Allen Hamilton Inc.
e-mail: paul.adamczyk@gmail.com

Fielding (2000), thus completing the process. HTML has followed a similar path. It started out with a simple set of tags for structuring text and graphics on Web pages. As the number of content types [new multimedia formats, more sophisticated ways of displaying text, interactive Web pages (Garrett 2005)] grew, the HTML tags were pressed into service of displaying them in various non-standard ways. After nearly two decades of this growth, new multimedia HTML tags are finally going to be added and standardized by W3C in HTML5, which is expected to be completed in 2012 (Hickson 2010).

A similar sequence of events – simple beginnings leading to an unruly explosion followed by some type of organization – can be observed in the realm of Web services. The first Web services were built for passing remote procedure calls (RPCs) over the Web. The idea took off quickly and resulted in a large collection of standards (beginning with SOAP and WSDL). Surprisingly, these standards were defined with little consideration for the contemporary practice; sometimes before there were any implementations to standardize. The end result of this premature standardization was confusion, rather than order that standards usually bring. In response, an alternative style of Web services, built according to the rules of the Web, began to appear. These (so-called RESTful) Web services are maturing, or, more precisely: people are re-learning to use the tried-and-true standards of the Web and applying them when building Web services. As the two styles of Web services are used side-by-side, one hopes that they will begin to have positive effects on one another. Currently, the interactions and comparisons begin to reach a constructive stage, so this is a good time to stop and reflect on the current state of affairs.

In particular, this chapter focuses on the interpretation of the widely used term, REST. Roy Fielding coined the term and codified it under four principles. In practice, people are implementing it in many ways, each harboring certain implicit conventions of the developers. Following the path of practice dictating the standards, we raise questions about the previously accepted views about REST and Web services, and identify the challenges raised by the current state of practice.

Having a standard meaning of RESTfulness would engage the enterprise community. REST has been an important part of "renegade" Web services, appealing more to independent, small-scale and "hip" developers. With concerted research effort, it would fulfill the stricter requirements of enterprise Web services; conversely, the enterprise services would benefit from its simplicity.

We begin by summarizing the theory behind RESTful Web services, and draw a comparison with WS–∗ services. Next, we look into the usage patterns of Web services in practice: both RESTful services and WS–∗ services. Then, we discuss some of the problems facing the existing RESTful services, how these problems make it harder to apply RESTful services to large enterprise systems, and how tools for implementing them help to alleviate these problems. We conclude by surveying some of the outstanding research problems of RESTful Web services.

Conventions used in this chapter. We consider two dominant styles of Web services: RESTful and WS–∗. The term Representational State Transfer (REST) was coined by Roy Fielding to identify an architectural style based on a set of principles for designing network-based software architectures (Fielding 2000). Subsequently, the term was extended to describe a style of building Web services based on the principles of REST. We use the term *RESTful* to refer to the Web services built according to this architectural style (or parts of it). We use term WS–∗ to refer to services based on SOAP, WSDL and other WS–∗ standards (e.g. WS-Addressing, WS-Security), which were defined specifically for Web services.

Web Services in Theory

Although this task was undertaken many times before, presenting a fair comparison of WS–∗ and RESTful Web services remains a daunting task. In this section, we will describe their guiding principles and summarize two studies that compare these architectural styles.

Principles

Roy Fielding documented REST based on the principles that emerged as the Web evolved (Fielding 2000). He noticed that Web servers, clients, and intermediaries shared some principles that gave them extensibility to work on the large-scale of the Internet. He identified four principles of REST (which he called constraints) (Fielding 2000):

1. Identification of resources.
2. Manipulation of resources through representations.
3. Self-descriptive messages.
4. Hypermedia as the engine of application state (abbreviated HATEOAS).

These principles describe the architecture of systems and interactions that make up the Web. The building blocks of the Web are called *resources*. A resource is anything that can be named as a target of hypertext (e.g., a file, a script, a collection of resources). In response to a request for a resource, the client receives a *representation* of that resource, which may have a different format than the resource owned by the server. Resources are manipulated via *messages* that have standard meanings; on the Web, these messages are the HTTP methods. The fourth principle means that the state of any client–server interaction is kept in the *hypermedia* they exchange, i.e., links, or URIs. Any state information is passed between the client and the server in each message, thus keeping them both stateless. It's easy to check any design against such a simple description. Any discrepancies will be easy to identify. However, this simplicity is deceptive – if one tries to simplify it even more, the entire design suffers. We will discuss concrete examples of oversimplifying REST in some Web services in "REST Concepts in Practice".

WS–∗ services do not have a single metaphor. Web Services Architecture document (W3C Working Group Note 2011) from W3C describes four architectural models of WS–∗, but does not explain how they relate. One of the models is the Resource Oriented Model (which would imply REST), but as their definition of Web services suggests, the systems they consider are limited to various standards: SOAP, WSDL, and others. New capabilities are added to WS–∗ in the form of new standards. There is no overarching description of the relationship between WS–∗ standards. Their definitions are constrained only by the compliance with SOAP, WSDL, and the XML schema for defining additional "stickers" in the SOAP envelope.

Comparison Between REST and WS–∗ Principles

Pautasso et al. study. In the most comprehensive comparison to date, Pautasso et al. (2008) compare RESTful and WS–∗ services on three levels: (1) architectural principles, (2) conceptual decisions, and (3) technology decisions.

On the level of *architectural principles*, Pautasso et al. analyze three principles (protocol layering, dealing with heterogeneity, and loose coupling) and note that both styles support these three principles. However, they can identify only one aspect common to both styles – loose coupling to location (or dynamic late binding). Consequently, they conclude that it's not possible to make a decision at this level and proceed with more detailed analysis. At the level of *conceptual decisions*, they compare nine different decisions and find that RESTful services require the designer to make eight of them, vs. only five for WS–∗. However, WS–∗ have many more alternatives than RESTful services. Finally, in the *technology* comparison, they identify ten technologies that are relevant to both styles. In this comparison, WS–∗ once again offer many more alternatives than their RESTful counterparts.

Based on these results, the authors recommend using REST for ad hoc integration and using WS–∗ for enterprise-level application integration where transactions, reliability, and message-level security are critical.

This study illustrates two key difficulties of performing convincing comparisons of broad ideas, such as Web service styles. First, it's difficult to select the most relevant principles to compare. Second, once the principles are selected, it's difficult to identify choices that are shared by the competing ideas.

Pautasso et al. do not explain why they selected protocol layering, dealing with heterogeneity, and loose coupling as the only architectural principles to compare. One would expect a comparison of principles to involve non-functional requirements (Bass et al. 2002) relevant to Web services. However, in their analysis, key -ilities (security, reliability) are only mentioned at lowest level of comparison, the technology decisions. Moreover, they shy away from comparing concepts that are relevant at the enterprise level (transactions, reliability, message-level security), even though they cite these very concepts in their concluding recommendation.

The actual comparison has two problems. First, they use the *numbers* of architectural decisions and available alternatives to choose which style is better. But counting is hardly the right metric – not every decision point has the same weight. Second, most decision points on every level have two options, one for each style, indicating that they actually have nothing in common. Only in a few cases do both styles require a decision on the same question. Nevertheless, this paper is the best-conducted comparison of principles available today. It's unbiased, thoroughly researched, and it examines multiple points of view.

Richardson and Ruby book. A second comparison of note is presented in the book, "RESTful Web Services" (Richardson and Ruby 2007). The authors, Richardson and Ruby, discuss the principles that are relevant to all systems available on the Web. Even though their book is biased toward RESTful Web services, the principles they discuss would be a better starting point for making a fair comparison between the two styles.

They identify four system properties of RESTful services: (1) uniform interface, (2) addressability, (3) statelessness, and (4) connectedness. In RESTful Web services, these properties are embodied in resources, URIs, representations, and the links between them. Lets consider how these principles apply to WS–∗ services. Addressability and some form of connectedness are embedded in the WSDL definition of bindings and ports. Many WS-*services are stateless (although it is not an explicit requirement). Having a uniform interface shared by all services is the only property not supported by WS–∗. Thus, WS–∗ services exhibits three of these four properties. WS–∗ services achieve these properties via different means, but these properties are clearly relevant to both, and therefore a good choice for comparison.

Richardson and Ruby use a similar approach to evaluate how RESTful Web services offer capabilities which are important for enterprise-level integration. They show how to implement transactions, reliability, message-level security (concepts that Pautasso et al mention, but do not discuss) using REST. We will return to these three concepts in "Ready for the Enterprise?".

Both styles of Web services possess certain characteristics that guide their design and development, although they are defined in ways that make it difficult to compare them side-by-side. Next, we will look at how services are used in practice, which provides yet another perspective for comparing them.

Survey of Existing Web Services

One obstacle to studying existing Web services is the fact that many of them are not accessible to the outside world, because they are proprietary. Proprietary systems have different requirements (fewer security threats due to well known vulnerabilities, no need to adhere to common standards) that result in different choices of Web services technologies. Industry studies provide some insight about the trends in proprietary Web services, such as the planned and actual usage of Web services. One industry survey shows that the adoption of SOAP standard by enterprises increased 31% between 2002 and 2003 (Correia and Cantara 2003). A follow-up survey from 2006 notes that about 12% of enterprises report completing a "full enterprise roll-out" and another 21% are in process, while 60% are still studying the feasibility of such projects (McKendrick 2011). Both surveys report only on WS–∗ Web services.

More recent results show a new trend. According to a 2008 Gartner Survey (Sholler 2008) there has been an increase in the number of organizations implementing Web services using Representational State Transfer (REST) and Plain Old XML (POX). RESTful Web services are considered less complex, require fewer skills, and have a lower entry cost than WS–∗ Web services. However, the surveyors believe that RESTful services by themselves do not provide a complete enterprise solution.

Turning our attention to public Web services, two earliest surveys of public Web services (Kim and Rosu 2004; Fan and Kambhampati 2005), from 2004, discussed strictly WS–∗ services. Both surveys showed that some of WS–∗ standards (most notably SOAP and WSDL) were successfully used in practice, but they did not cover other standards. These surveys have been limited to WS–∗ services, perhaps unintentionally, because they considered the presence of a WSDL file as a necessary prerequisite of a valid Web service.

In order to build on their work, we have studied various Web services repositories (including the only extant ones cited by these surveys) to analyze the available public Web services from the perspective of architectural styles they follow. We performed these surveys in mid-2007 and again in mid-2010 by examining the Web services listed in the following repositories:

- `xmethods.net`
- `webservicex.net`
- `webservicelist.com`
- `programmableweb.com`

These repositories describe only publicly accessible Web services. While SOAP services are easy to find automatically (by checking for the presence of the WSDL file), RESTful services are documented in non-standard ways that make their automatic discovery impossible. We examined the type of each service manually, by reading its documentation. We have identified four mutually exclusive categories of Web service styles: RESTful, WS–∗, XML-RPC, and Other. XML-RPC was the first attempt at encoding RPC calls in XML (which later evolved into SOAP). The Other category groups many other types of services, including RSS feeds, Atom, XMPP, GData, mail transfer protocols. The most popular styles of Web services in each repository are shown in Table 2.1.

At a first glance, these results could not possibly paint a more inconsistent picture. Each repository shows a different trend. However, the differences arise from the nature/focus of these repositories. The first two repositories, which list (almost) exclusively WS–∗ services, advertise services that require payment for access. The

Table 2.1 Web service styles used in public services

	xmethods		webservicex		webservicelist		programmableweb	
Style	2007	2010	2007	2010	2007	2010	2007	2010
RESTful	3	0	0	0	103	144	180	1627
WS–∗	514	382	71	70	233	259	101	368
XML-RPC	1	0	0	0	6	21	24	53
Other	0	0	0	0	98	35	90	207
Total	518	382	71	70	430	459	395	2255
(unique)	(514)				(411)	(386)	(340)	(2179)

Survey conducted in 2007 and 2010. Some service are available in two or more styles. The number of unique services is shown in parentheses

second repository appears to be closed to registration (we could not find any way to contact the owners to register a new service) which may imply that they are advertising only the services which they own. The numbers of services listed in these two repositories have not changed much in the last 3 years.

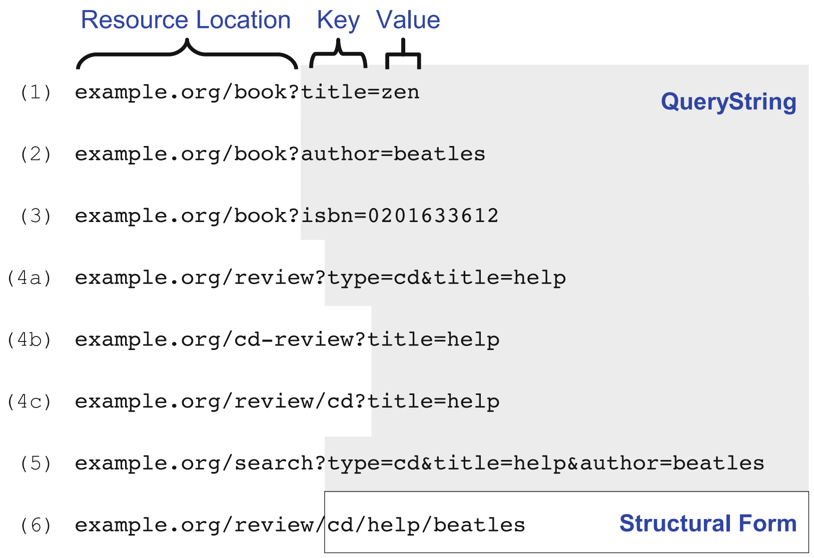
The latter two repositories feature a variety of Web service styles, with RESTful and WS–∗ services being the two most popular styles in both the 2007 and 2010 tally. Programmableweb.com is the only repository that shows an increase in the number of services; a fivefold increase over the observed period. Its data shows increase in all types of services, but mostly in RESTful ones, which currently account for about 75% of services listed, compared to less than 50% 3 years earlier.

These results, although insufficient to determine conclusively which style is more popular (and why), indicate that a wide variety of public Web services is available and that a sizable number of RESTful services has been created recently, even if not all of them are widely known.

REST Concepts in Practice

With so many public Web services available to study, we were able to identify many trends in how closely services follow the theoretical principles of REST. WS–∗ principles are encoded in XML-based standards that are easy to enforce by tools. The designer selects the necessary features (standards), then finds the tool that supports them. The actual development is easy. But since this book is about REST, we will focus on RESTful Web services, and refer to WS–∗ only to compare and contrast specific features. In this section, we will review how REST principles are embodied and implemented in actual RESTful Web services.

According to the principles of REST, which we introduced in "Web Services in Theory", every resource is identified with a URI. In response to HTTP messages, resources return their representations to clients, or the clients modify the resources.



```
(1)  example.org/book?title=zen
(2)  example.org/book?author=beatles
(3)  example.org/book?isbn=0201633612
(4a) example.org/review?type=cd&title=help
(4b) example.org/cd-review?title=help
(4c) example.org/review/cd?title=help
(5)  example.org/search?type=cd&title=help&author=beatles
(6)  example.org/review/cd/help/beatles
```

Fig. 2.1 Examples of RESTful hypermedia defined as URIs. Examples 1–5 use query strings of form `key=value`. Examples 4a–c show alternative ways to define the same resource. Example 6 uses the structural form instead of query strings: the order of keywords is defined by the server's API so that the client need not list keys, only values, in the URI

Proponents of RESTful Web services typically say that every service needs to follow the CRUD model (Kilov 1990). This concept, borrowed from the database domain, defines one method for creating, reading, updating, and deleting a resource on the server (corresponding to POST, GET, PUT, and DELETE methods). This approach enables invoking different operations on a resource by applying a different HTTP method. This is only possible if resources are defined in a correct way. Figure 2.1 shows some examples of valid URIs. All of these URIs can be accessed with the GET method.

One good example of a Web service that follows the principles of REST is Amazon S3 (Simple Storage Service) (Amazon 2011). S3 defines many resources and uses HTTP methods (POST, GET, PUT, DELETE, even HEAD) for manipulating them. It uses HTTP error codes correctly and shows how to map various errors to HTTP codes (the API references 13 unique HTTP status codes in the 300–500 range). S3 also supports caching by including ETag header that clients can use in conditional GET.

However, most RESTful services are not designed as diligently. They neglect to follow the principles in various ways. In order to evaluate the current level of understanding of REST, we will look at some representative mistakes from the perspective of the 4 principles of REST.

Identification of Resources

Every designer of a RESTful service must answer the question: What constitutes the resources of the system? Ideally, any concept within the system that has a representation should be exposed as a resource.

In WS–∗ services, clients invoke API methods on the server by passing SOAP messages to a well-known service end-point defined with a URI. These service end-points are the only resources used by WS–∗. Some RESTful service follow the same pattern – they define one path component to be used in every URI and encode parameters for the corresponding server method in the query strings. This is wrong, because in REST resources are supposed to be accessed with self-descriptive messages (e.g. HTTP methods) that have well-defined semantics. Looking at Example 5 in Fig. 2.1, it's OK to access this resource via GET, but what would be the intended semantics for PUT and other HTTP methods? Such a resource can only accept read-only requests, the way Google's search service works. But if the clients need to be able to modify the resources, this style of resources is not appropriate.

Defining resources is hard. Consider, for example, a hypothetical Web service that provides information about books and music. Such a service should define multiple resources, `book`, `cd`, `review` that are queried by title, author, or ISBN. Example 1 in Fig. 2.1, `example.org/book?title=zen` represents a resource for books that contain "zen" in the title. Examples 2 and 3 show how to query the resource by author and ISBN. There are several options for defining resources corresponding to a review. The system could have one review resource (as in 4a), a dedicated resource for each product type (in 4b), or a composite resource (review) with individual children resources, one per product type (4c). These all are valid choices. Alternatively, as in Example 6, the URI structure can enforce a specific order of parameters (type, then title, then author), thus making it unnecessary to specify the type of each sub-element in the URI. Note that this format requires implicit understanding of the structure of this URI, which is defined outside of the URI by the provider of this Web service.

The problem of designing resources is similar to teaching object-oriented design to programmers, who were first taught procedural languages – it requires a changed mindset. One can define resources without deep understanding of REST, but it's unlikely that such design will take full advantage of all available features of HTTP and URI standards as objects/classes. In the second step, the public methods of the object are defined. In any non-trivial problem, these two steps identify many objects and many methods. The application is built by connecting the objects, which invoke methods on one another. A similar approach can be applied to defining resources, except that only the first step identifies many objects (i.e. resources). The available HTTP methods are defined in the standard and links between resources are traversed at run-time. Thus steps 2 and 3 come for free in HTTP, but only if step 1 is done well.

Representations

If resources support multiple representations, they can produce responses in different data formats. In HTTP, clients specify their preferred formats in `Accept-*` headers for content negotiation. By conforming to HTTP, RESTful Web services can support multiple types of response (MIME) formats, just like the Web does, which makes it easy to comply with this principle.

Many RESTful Web services support at least two response formats (typically XML and JSON). Library of Congress Subject Headings Web service is the only service listed at programmableweb.com that advertises the support of content negotiation. It serves content in four different types (XHTML with embedded RDFa, JSON, RDF/XML, and N3). Unfortunately, other services do not appear to support this important feature of HTTP, because we did not find it in their documentation.

Self-descriptive Messages

REST constrains messages exchanged by components to have self-descriptive (i.e. standard) definitions in order to support processing of interactions by intermediaries (proxies, gateways). Even though HTTP/1.1 defines eight methods, only two of them, GET and POST, have been used extensively on the Web, in part because these were the only methods supported by the early Web browsers.

Early RESTful Web services show difficulties in understanding the differences between even these two methods.[1] Some services defined GET for sending all requests to resources, even if the requests had side effects. For example, initially, Bloglines, Flickr, and Delicious Web services defined GET for making updates to these services (Dare Obsanjo Blog 2011). Other services specified that clients can use GET and POST interchangeably, which is equally wrong. Consequently, these services were misusing Web proxies and caches polluting them with non-cacheable content, because these Web systems rely on standard meanings of HTTP methods. Since then, the offending APIs were modified, but the underlying problem of understanding the semantics of HTTP methods still remains.

Many RESTful proponents consider the use of 4 HTTP methods corresponding to CRUD operations as a sign of good RESTful design. But these methods are not sufficient to express complex operations on resources. They provide only simple data-access operations. These methods need to be combined into sequences in order to implement even the simplest transactions.[2] That's why many RESTful services try

[1]GET sends data from the server to the client, in the response. POST sends the data from the client to the server, in the request. Thus, GET is used for reading, and POST for writing.

[2]A simple bank transaction, e.g. transferring $100 from savings to checking, involves sending four HTTP requests. First, create a resource for the transfer using POST. Next, send a PUT to the

to encode more complex operations (such as "search") into URIs in RPC style even though they know that it violates REST. Another reason why the CRUD metaphor is not a good match is that HTTP methods POST and PUT do not map exactly to CRUD's "create" and "update," respectively. PUT carries a representation produced by the client, which the server should use to replace its contents (so it serves as both create and update). POST means the server decides how to use the representation submitted by the client in order to update its resource.

This problem of not taking full advantage of HTTP methods is not unique to Web services. Typical Web applications (accessible via browsers) use only two HTTP methods in practice. In a study of HTTP compliance of Web servers (Adamczyk et al. 2007), we found that Web servers and intermediaries understand correctly only GET and POST methods. Only a fraction of popular websites send compliant responses to other HTTP methods, even though the popular Web servers implement all these methods correctly. These compliance results haven't changed much since HTTP/1.1 standard has been released, in 1999.

The inclusion of the 4 HTTP methods corresponding to CRUD operations in a definition of a RESTful service is only a first step in satisfying the principle of self-describing messages. This principle means that methods should be used according to their standard definitions. A case in point is the new HTTP method, PATCH, added in March 2010 (Dusseault and Snell 2010). It is intended to complement PUT and replace some uses of POST with more precise semantics. With POST, the client cannot specify *how* the resource is to be updated. Unfortunately, the definition of PATCH does not define the structure for including the instructions to update (i.e., patch) the resource. A standard definition of the instructions will be necessary to make this method interoperable. As the additions of PATCH indicates, the set of *relevant* HTTP methods is not static. The WebDAV protocol (which RESTful proponents tend to overlook) defines 8 more methods for distributed authoring and manipulating collections of resources (Goland et al. 1999). Thus RESTful Web services have many self-describing methods to choose from. Although today most Web services don't use their HTTP methods right, we hope that in time they will.

HATEOAS

Hypermedia as the engine of application state means that neither client nor server needs to keep the state of the exchange in a session, because all the necessary information is stored in the exchanged HTTP messages (in the URI and the accompanying HTTP headers and body). Defining self-contained links is critical for RESTful Web services, because these links make it possible to traverse, discover, and connect to other services and applications.

resource specifying the withdrawal of $100 from savings. Then, send a second PUT to deposit $100 to checking. Finally, send a PUT to commit the transaction. Note that the burden of verifying that each step was successful is on the client. If a step fails, the client needs to send a DELETE to the transaction resource to abort the transaction (Richardson and Ruby 2007).

However, this is difficult, because complex interactions translate to complex URIs. Large applications have many states that the client needs to be aware of. HATEOAS forces Web services to expose these states as links, which appear to duplicate the internal implementation of the service. To avoid this duplication, some RESTful Web services resort to exposing the underlying API of the service even if they know it's wrong.

Many services require the client to send user-specific information (e.g. user-id) in every request URI. As a result, the same requests from two different clients appear unique to the Web caches, because caches use URIs as keys for the data. Sending user-specific information is often unnecessary (especially when the user sends a generic query), but it's used extensively by Web services providers in order to limit the number of accesses from each client. Since HTTP caching cannot be used in this case (except when the same user requests the same resource again), the service must handle more requests, which defeats the purpose of rate limiting. This seemingly innocuous (but often occurring) lapse violates two principles – the identification of resources and HATEOAS because the URIs representing states cannot be used by other users. It also affects cacheability.

Other Important Concepts

The HTTP standard defines the meaning of different error conditions and several mechanisms for caching. Compliant RESTful Web services should follow them.

Initially, RESTful services copied their error-handling mechanism from SOAP. Many Web services would not use HTTP status codes (e.g. "404 Not Found") to describe the result of a request, but rather always returns "200 OK" with the actual status is hidden in the response body. Other services (e.g. earlier versions of Yahoo Web services) defined their own status codes that were incompatible with the standard ones. By using service-specific codes, they would not take advantage of existing Web systems that understand these codes thus forcing clients to build specialized, non-interoperable software to handle them. Fortunately, most Web services we surveyed now do use HTTP status codes, and only add service-specific extensions for new statuses. For example, Delicious uses codes 500 and 999 to indicate that user request was throttled (due to exceeding a pre-defined limit of connections). HTTP does not have a status that corresponds to this condition, so it makes sense to define a new one.

Our survey gathered little information about caching. Aside from exceptional Web services like S3 (and even they don't use the term caching in the documentation), RESTful services do not document if they support caching. Of course, the services that employ user-ids could not benefit from caching anyway.

As the length of this section indicates, RESTful services still have difficulty in following the principles of REST. There are few fully compliant service definitions, but it's easy to find examples of services that violate any of the principles. On the bright side, we have observed a lot of improvements in compliance over the last few years. RESTful services, by the virtue of being public are more open to general scrutiny. Users can discuss the design decisions in the open, criticize them, and see changes in the next version. To gain a better perspective of the positive changes that occurred over the years, the reader is encouraged to browse the discussion of these and other violations documented at RESTWiki (2011).

An important question is: Why are many services that attempt to be RESTful not compliant with the principles of REST? Are these principles too restrictive? Too hard to implement? Unnecessary for Web services (as opposed to Web pages and Web applications)? It's still too early to tell.

Frameworks for Building RESTful Web Services

The improvements in the understanding of the principles of REST, as indicated by the slow but steady elimination of bad design decisions from public RESTful Web services can be attributed to software tools and frameworks that have began to appear in the last few years.

Support of REST Principles

Many frameworks and tools for building RESTful Web services are available today. They are written in different programming languages and range from simple to quite sophisticated in their support of HTTP and other Web technologies. As they continue to improve, misunderstandings and violations present in today's Web services will likely lessen.

We have examined ten popular frameworks that provide automated support for building software according to the principles of REST. Some frameworks, like Ruby on Rails and Spring are generic Web frameworks, while others are specific to RESTful services. Table 2.2 summarizes key features of these frameworks, grouped by REST principles. The frameworks are listed alphabetically, sorted by the programming language and name. The second column in the table shows how these frameworks support defining resources (corresponding to REST principles 1 and 4). Almost all the frameworks provide some support for building resources (URIs) and hyperlinks – through URI templates (Gregorio et al. 2010), annotations in the target programming language, or other types of mappings. The third column shows which types of multimedia are supported and how (principle 3). Most frameworks enable generation of multiple representation formats. The fourth column shows which HTTP methods are supported (principle 4). Most of them support GET, POST, PUT, and DELETE HTTP methods, either directly, or by specifying the desired method in an auxiliary parameter [such as the `X-HTTP-Method_Override` header, or the hidden "_method" form field (Richardson and Ruby 2007)]. The last column points out other interesting features provided by the frameworks. Few brave frameworks have ventured into implementing more advanced concepts of caching, automated testing, or authentication.

Table 2.2 How frameworks for building RESTful Web services support the principles of REST

Name (Prog. language)	Resources and HATEOAS	Representation	Messages	Other (API, caching, status codes, etc.)
Jersey (Java)	Annotations for URI mappings	MIME types, XML, JSON and Atom	GET POST PUT DELETE	Support for JAX-RS. Testing framework
RESTEasy (Java)	Annotations for URI translations and variable mapping	Annotations for output representations (many types supported). Content negotiation	GET POST PUT DELETE	Output caching and compression Support for JAX-RS
Restlet (Java)	URI templates and variable binding	Supports various output representations	GET POST PUT DELETE	Support for JAX-RS. Caching headers set in `Conditions` class. Security checks added via filters. All HTTP status codes
Spring (Java)	Templated URIs using Java annotations	Content negotiation with Accept header or by URL inspection (read file extension)	GET and POST directly, PUT and DELETE with `_method`	ETag header for caching
Recess (PHP)	URI templates and variable extraction using annotations	Not supported	GET POST PUT DELETE	Not supported
Routes (Python)	Proper URL syntax; No IDs in query parameters	Not supported	GET POST PUT DELETE	Not supported

CherryPy (Python)	Simple mapping: HTML forms to Python variables	HTML forms	GET and POST	An object tree generated to map requests to Python functions
Django (Python)	URI templates for mapping advanced URL patterns to Python code	Targeted output formats: XML, JSON, YAML	GET POST PUT DELETE	Caching. HTTP status codes supported by Python libraries
RESTfulie (Ruby, Java)	emphasizes hypermedia links	many formats; content negotiation	GET POST PUT DELETE	HTTP status codes; integrates with Ruby on Rails
Ruby on Rails (Ruby)	Route configs map URI to `Component` class (imposes URI conventions)	Excellent support of many data formats – e.g. Accept header	GET POST PUT DELETE	Conditional GET for caching RESTful authentication

Several of the Java frameworks support JAX-RS, a Java API for RESTful Web services. They are Jersey (considered the reference implementation), Restlet, and RESTEasy. JAX-RS specifies how to map Java classes to Web resources using Java annotations. The annotations specify the relative path of the resource (part of the URI) for a Java class, which Java methods correspond to HTTP methods, which media types are accepted by the class, and how to map class properties to selected HTTP headers (Hadley and Sandoz 2009).

Aside from Django, all the Python and PHP frameworks offer only rudimentary support for REST. Other frameworks include more advanced features, but they still fall short of supporting all principles of REST. Most frameworks define schemes for mapping URIs to classes and methods, but not all of them are as flexible as HTTP requires, e.g. Ruby on Rails imposes constraints on URI formats. Only one framework (Restlet) supports all HTTP status codes. No framework supports all flavors of HTTP caching, and many do not support caching at all.

The principle of HATEOAS (unambiguous semantics for following and embedding links) is not well supported. Only the RESTfulie framework emphasizes the importance of this principle. Let's consider a simple example of the expected behavior. When a client requests a resource (e.g. information about a collection of items) it should be easy to construct a URI to refer to an individual item from that collection. Frameworks should provide built-in support for such conversions of URIs. Currently, this mapping work must be implemented manually in the client code, because most frameworks do not support it.

Overall, the RESTful frameworks need to include more functionality to be fully compliant with REST. But the biggest problem is that even if they do implement the support for a principle, the frameworks have no mechanisms to enforce that it is applied correctly in the client code.

Ready for the Enterprise?

Frameworks make it possible to build bigger Web services, and their capabilities keep on growing. Is that enough to persuade enterprise system architects to switch to RESTful Web services? Recall the study of Web services by Pautasso et al. (2008) we discussed in "Web Services in Theory". They cite security, reliable messaging and transactions as key differentiators between RESTful and WS–$*$ services. To be ready for enterprise, RESTful frameworks need to support these features. Richardson and Ruby (2007) show how these concepts can be implemented using HTTP.

For basic message-level security, it's enough to use HTTPS. But more complex capabilities such as signatures, encryption, or federation (enabling a third party to broker trust of identities) cannot be supplied by HTTP alone. Further research is required to define these concepts properly in RESTful Web services (more about this in "Open Research Problems of RESTful Services".)

To provide reliable messaging, one needs to ensure that all HTTP methods are idempotent. This property makes it possible to replay any method, as necessary, to make sure that it succeeded. Of course, this approach to reliable messaging is tedious and currently requires a lot of manual coding on the client side.

Implementing transactions with HTTP messages requires exchanging many messages, which can get complex quickly (as we saw in "Self-descriptive Messages"). Current frameworks are not mature enough to abstract out/encapsulate common transaction patterns. But transactions are needed as building blocks of workflows, which occur often in enterprise systems. A proposed extension to the Jersey framework introduces *action resources* for specifying workflows (Hadley et al. 2010). Each action resource exposes one workflow operation available on the service. The client obtains the workflow specification (i.e. the list of action resources) at the beginning of the sequence. In line with the principle of HATEOAS, it's the client's responsibility to keep track of the current state of the system throughout the execution in order to invoke the workflow resources in the correct order. This is a dynamic approach, because the exact sequence of the workflow need not be specified until the client begins to execute it.

But even if security, reliable messaging, and transactions are solved successfully, RESTful services must also demonstrate scalability. Compared to large legacy systems on top of which many WS–$*$ services are built, current RESTful services are small. Tool support is needed for combining disparate services to build larger ones and for automating the generation of URI schemas that can adapt when a service is being extended.

Today's frameworks are not yet ready to support enterprise needs. They do not implement advanced security features or transactions; they do not verify that HTTP methods they generate are idempotent, which is the necessary prerequisite for reliable messaging; they are not scalable. Implementing these features is a matter of time, because HTTP already defines most of the necessary concepts to perform these tasks. However, it's not enough that the frameworks implement the necessary functionality. The frameworks must guide and force the users to recognize the correct features for the job and to apply them correctly.

Open Research Problems of RESTful Services

REST originated at the intersection of academia and software development, among the architects of the World Wide Web. Fielding's research culminated in authoritative versions of HTTP and URI standards that define the unique characteristics of the Web. Unfortunately, researchers have only recently started to work on RESTful services. As late as 2007, there were no papers about RESTful Web services in either ICWS, ECOWS, or WWW conferences. In 2010, ICWS has featured several papers about RESTful services and the WWW conference has hosted the first "Workshop on RESTful Design (WS-REST 2010)" (Pautasso et al. 2010), which is a welcome sign.

Proponents of RESTful Web services made their first attempts to reach the research community via conference presentations (Prescod 2002; Haas 2005), and computer magazine editorials (Vinoski 2008). Recently, survey papers (Pautasso et al. 2008), and new research work (Pautasso et al. 2010; Overdick 2007) began to appear. Hopefully, this book will advance the state of research even farther.

The problems we discuss below are concerned with non-functional requirements and how they can be supported by RESTful services. Many of these research efforts are defining new Web standards. Web linking (Nottingham 2010) aims to improve cache invalidation. HTTP PATCH (Dusseault and Snell 2010) defines a new method to make more maintainable services. URI templates (Gregorio et al. 2010) make it easier to define groups of resources with regular expressions. OAuth (Hammer-Lahav 2010) secures authentication and data sharing in HTTP-based systems.

Caching

Of many aspects of performance, caching is one of the best examples of why it pays to use HTTP correctly. The data may be cached by the client, by the server, or by intermediaries, such as Web proxies. In the early days of mostly static content, 24–45% of typical Web traffic was cacheable (Duska et al. 1997). Today, the estimated range is 20–30% (Nottingham 2009), which is very impressive considering how dynamic the Web content is.

Unfortunately, most of the RESTful services aren't benefiting from caching: many frameworks don't support caching, and typical URIs are not cache-friendly, because RESTful Web services require user info in each request. We have already discussed how user-ids are used for rate-limiting, in "HATEOAS". It is unlikely that Web services will ever change this policy. Instead, it would be better to move user-specific information out of the URIs, so that the responses can still be cacheable.

An upcoming addition of Web Linking (Nottingham 2010) (for improving cache invalidation) indicates that the HTTP community values caching. However, it's very difficult to keep up with all the variations: caching headers, tags, expirations, and conditional methods. Caching is so complex that even the upcoming HTTPbis specification from IETF divides this topic into two documents (Caching proper and Conditional Requests). Caches are not unique to the Web: caching in computer architecture is understood well. We are lacking a single, consistent model of caching on the Web.

Maintainability

Typical maintenance tasks of Web services (adding new features, fixing service APIs) affect services themselves, their documentation, the client code, and even the development tools. Since RESTful Web services are still prone to wholesale changes, each of these facets offers ample opportunities for research.

Changes of Web service definitions necessitate upgrades of the client code. When a new version of a service becomes available, clients need to adapt their code. Neither WS–∗ nor RESTful services providers are concerned with making client updates easier. They claim that there is no need to deprecate APIs, because they will always be available, so clients are not required to upgrade. Ideally, this would be the case, because well-named resources do not need to change (Berners-Lee 2011). New Web services might be able to preserve their APIs for some time, but maintaining several versions isn't realistic if a service plans to grow. Some services offer software development tools for building client applications, but they suffer from the same types of challenges as typical software – APIs change. Is it time to start exploring refactoring of Web service APIs?

Security and Privacy

Securing RESTful Web services is a multi-faceted endeavor: it involves securing the data, as well as the entire communication. One must protect the confidentiality and integrity of data. The data in transit should be filtered for malicious payload. The communication should support authentication and access control, and ensure that the privacy of the communicating parties is not compromised.

Compared to the WS-Security framework (Web Service Security WSS), RESTful services rely on various add-ons that work on top of HTTP. HTTPS (Rescorla 2000) is widely used for confidentiality, but it only provides hop-by-hop security. Developers should adopt message level security mechanisms. Unlike WS–∗, there are no standards to follow, but practitioners follow various reference architectures, e.g. Amazon S3 service (Amazon 2011). Amazon S3 also incorporates timestamps to guard against request replaying. Various client side and server side filters should be employed to validate the content.

HTTP supports basic and digest-based authentication mechanisms (Franks et al. 1999), but both have their weaknesses (Apache HTTP Server v2.2 2011). Current services delegate identity management and authentication mechanism to a third party, and rely on a claims-based authentication model. Technologies for supporting authentication for HTTP-based services are emerging, e.g. OpenId (Fitzpatrick 2005) for federated identity, and OAuth 1.0 (Hammer-Lahav 2010) for authentication and data sharing. These protocols open up new avenues of research. For example, OAuth is going through a revision in October 2010, where the protocol writers are considering dropping cryptographic operations and relying on SSL to protect plaintext exchange of authentication tokens. They are trading off end-to-end security for ease of programming, and this decision should be validated by research. Another emerging protocol is XAuth (Meebo Dev Blog 2010), an open platform for extending authenticated user services across the Web, but it still has a lot of open security problems.

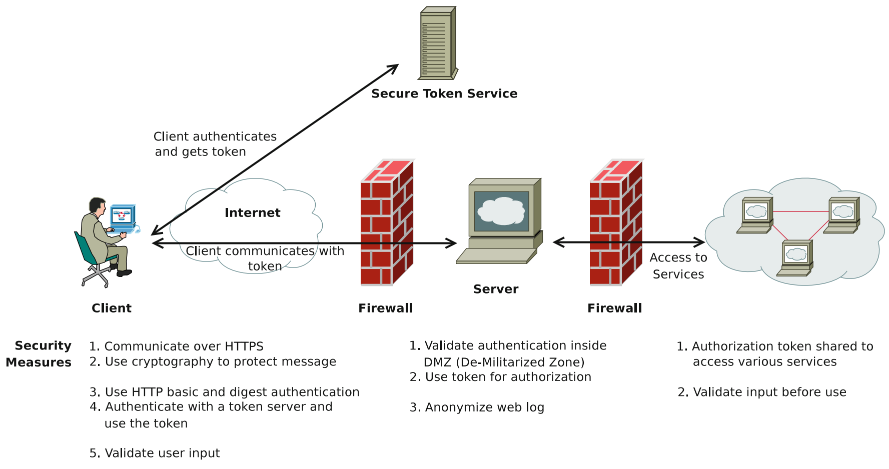

Secure Token Service
Client authenticates and gets token
Internet
Client communicates with token
Client
Firewall
Server
Firewall
Access to Services
Security Measures
1. Communicate over HTTPS
2. Use cryptography to protect message
3. Use HTTP basic and digest authentication
4. Authenticate with a token server and use the token
5. Validate user input
1. Validate authentication inside DMZ (De-Militarized Zone)
2. Use token for authorization
3. Anonymize web log
1. Authorization token shared to access various services
2. Validate input before use

Fig. 2.2 Security measures adopted at different layers in RESTful systems

Storing URIs in web logs may lead to privacy problems if the logs are not protected and anonymized. WS–∗ services do not store sensitive data in HTTP method signature and query strings. On the other hand, URIs created for RESTful Web services become the audit trail, and they should be anonymized.

Figure 2.2 illustrates a hypothetical model of how the security and privacy measures can be applied together. It shows a secure token service, a key entity in a third party authentication model. Note that the figure does not define the actual steps of an ideal protocol; it is an open research problem. Researchers also need to figure out how the security measures fit the REST model.

QoS

When multiple providers offer the same service, a client has a choice and can select the most suitable one. Often, this choice comes down to the Quality of Service (QoS) parameters. RESTful Web services today ignore QoS requirements; their only concern is providing functional interfaces. To add QoS parameters to RESTful services, a language for describing the parameters and a mechanism to incorporate the description in the HTTP payload is needed. Defining a standard QoS description language might benefit from the work in Semantic Web. Semantic Web ontologies define standard ways of interpreting information, such as QoS parameters, enabling all clients to interpret them the same way.

Studies of Existing Systems

Web services are good candidates for studying how software engineering concepts are followed in large, publicly available systems. But there have been few successful studies of RESTful services, or side-by-side comparisons of a service that exposes two interfaces defined in the competing styles (one RESTful, one WS–∗).

It is not easy to compare these two styles at the level of principles. The first order of research is to identify good principles for making the comparison. Zarras (2004) identifies the following principles for comparing middleware infrastructures: openness, scalability, performance, and distribution transparency. Properties of software architectures (Bass et al. 2002) is another source of principles to consider. Another possibility is to apply the same principled approach Fielding used to derive REST in order to define both RESTful and WS–∗ architectural styles. This would entail selecting and applying additional constraints, one at a time, to derive complete definitions of both architectural styles.

Conclusion

RESTful Web services (and Web services in general) pose the first serious test of the principles of REST, as identified by Fielding. On the one hand, the emergence of RESTful Web services, in response to WS–∗ services can serve as an indication that REST *is* the correct architecture for the Web. On the other hand, the state of practice still shows gaps in understanding and applying the theory behind REST, thus indicating that the process is not complete.

Up until a few years ago, there was a simple dichotomy between REST and WS–∗. RESTful services were used only for simple, public services. In contrast, enterprise standards, tools vendors, and the research community were only concerned with WS–∗ services. This is no longer the case – both styles are being used in all domains. The new challenge is to use them correctly, and to be able to align them to solve the real problems of the enterprise. Can RESTful services scale up to the enterprise-size challenges? We believe so. Amazon, Google, Yahoo, Microsoft, and other big companies have been building large, scalable, extensible, and relatively secure systems on the Web. RESTful services have the same basic principles to follow.

This concludes our whirlwind overview of how Web services relate to REST, in theory and in practice. Other chapters in this book will explore these topics in more details.

Acknowledgements The authors would like to thank Nicholas Chen, Hakan Erdogmus, Jim Webber, and anonymous reviewers for reviewing earlier drafts of this work.

References

L. Bass and P. Clementes and R. Kazman. *Software Architecture in Practice, 2nd Edition*. Addison Wesley, 2002.

P. Adamczyk, M. Hafiz, and R. Johnson. Non-compliant and Proud: A Case Study of HTTP Compliance, DCS-R-2935. Technical report, University of Illinois, 2007.

Amazon. Amazon Simple Storage Service API Reference, May 2011. http://docs.amazonwebservices.com/AmazonS3/latest/API/

Apache HTTP Server v2.2. Authentication, authorization and access control, May 2011. http://httpd.apache.org/docs/2.2/howto/auth.html.

T. Berners-Lee. Cool URIs don't change, May 2011. http://www.w3.org/Provider/Style/URI.html.

T. Berners-Lee, R. Fielding, and H. Frystyk. RFC 1945: Hypertext Transfer Protocol — HTTP/1.0, May 1996.

J. Correia and M. Cantara. Gartner sheds light on developer opps in web services. *Integration Developers News*, June 2003.

Dare Obsanjo Blog. Misunderstanding REST: A look at the Bloglines, del.icio.us and Flickr APIs, May 2011. http://www.25hoursaday.com/weblog/PermaLink.aspx?guid=7a2f3df2-83f7-471b-bbe6-2d8462060263.

B. M. Duska, D. Marwood, and M. J. Freeley. The measured access characteristics of World-Wide-Web client proxy caches. In *USENIX Symposium on Internet Technologies and Systems, USITS*, 1997.

L. Dusseault and J. Snell. RFC 5789: PATCH Method for HTTP, Mar. 2010.

J. Fan and S. Kambhampati. A Snapshot of Public Web Services. In *SIGMOD Record, Vol. 34, No. 1*, Mar. 2005.

R. Fielding. Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation. Technical report, University of California, Irvine, 2000.

R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext Transfer Protocol — HTTP/1.1, June 1999.

B. Fitzpatrick. OpenID, 2005. http://openid.net/.

J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. RFC 2617: HTTP Authentication: Basic and Digest Access Authentication, June 1999.

J. Garrett. Ajax: A New Approach to Web Applications, Feb. 2005. http://adaptivepath.com/ideas/essays/archives/000385.php.

Y. Goland, E. J. Whitehead, A. Faizi, S. Carter, and D. Jensen. HTTP Extensions for Distributed Authoring WebDAV. Internet proposed standard RFC 2518, Feb. 1999.

J. Gregorio, R. Fielding, M. Hadley, and M. Nottingham. URI Template (draft), Mar. 2010.

H. Haas. Reconciling Web services and REST services (Keynote Address). In *3rd IEEE European Conference on Web Services (ECOWS 2005)*, Nov. 2005.

M. Hadley, S. Pericas-Geertsen, and P. Sandoz. Exploring Hypermedia Support in Jersey. In *WS-REST 2010*, Apr. 2010.

M. Hadley and P. Sandoz. JAX-RS: Java API for RESTful Web Services (version 1.1), Sept. 2009.

E. Hammer-Lahav. RFC 5849: The OAuth 1.0 Protocol, Apr. 2010.

I. Hickson. HTML5: A vocabulary and associated APIs for HTML and XHTML, Oct. 2010.

Joe McKendrick. Service Oriented Blog, May 2011. http://www.zdnet.com/blog/service-oriented/?p0542.

H. Kilov. From semantic to object-oriented data modeling. In *First International Conference on Systems Integration*, pages 385–393, 1990.

S. M. Kim and M. Rosu. A Survey of Public Web Services. In *WWW 2004*, 2004.

L. Richardson and S. Ruby. *RESTful Web Services*. O'Reilly, Oct. 2007.

M. Nottingham. HTTP Status Report. In *QCon*, Apr. 2009.

Meebo Dev Blog. Introducing XAuth, Apr. 2010. http://blog.meebo.com/?p=2391.

M. Nottingham. Web Linking (draft), May 2010.

H. Overdick. Towards resource-oriented BPEL. In C. Pautasso and T. Gschwind, editors, *WEWST*, volume 313. CEUR-WS.org, 2007.

C. Pautasso, E. Wilde, and A. Marinos. First International Workshop on RESTful Design (WS-REST 2010), Apr. 2010.

C. Pautasso, O. Zimmermann, and F. Leymann. RESTful Web Services vs. "Big" Web Services: Making the Right Architectural Decision. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 805–814, New York, NY, USA, 2008. ACM.

P. Prescod. Roots of the REST/SOAP Debate. In *Extreme Markup Languages, EML*, 2002.

E. Rescorla. RFC 2818: HTTP over TLS, May 2000.

RESTWiki, May 2011. http://rest.blueoxen.net/cgi-bin/wiki.pl.

D. Sholler. 2008 SOA User Survey: Adoption Trends and Characteristics, Sept. 2008.

S. Vinoski. Serendipitous reuse. *IEEE Internet Computing*, 12(1):84–87, 2008.

W3C Working Group Note. Web Services Architecture, May 2011. http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.

Web Service Security (WSS). Web Services Security: SOAP Message Security 1.1, Feb. 2006.

A. Zarras. A comparison framework for middleware infrastructures. *Journal of Object Technology*, 3(5):103–123, 2004.

Part II
Design

Chapter 3
RESTful Domain Application Protocols

Ian Robinson

Abstract This chapter discusses the significance of domain application protocols in distributed application design and development. Describing an application as an instance of the execution of a domain application protocol, it shows how we can design RESTful APIs that allow clients to drive the execution of a domain application protocol without binding to the protocol itself. The second half of the chapter provides a step-by-step example of a RESTful procurement application; this application realizes a procurement protocol in a way that requires clients to couple simply to media types and link relations, rather than to the protocol.

Introduction

This chapter reflects the concerns of systems architects and developers charged with satisfying specific business needs – with getting things done. REST's hypermedia constraint (Fielding 2000) is all about getting things done: at the heart of the constraint is a compact of application, application protocol and application state that addresses the need to do useful things with computerized behaviors, to effect the kinds of changes in application state that release value to the providers and consumers of a business capability.

From an analytical perspective, every useful application of computerized behavior can be said to evidence what I call an underlying *domain application protocol* – much as every meaningful utterance evidences an underlying natural-language grammar. The design strategies I present in this chapter represent acts of deliberate discovery through which we come to understand the domain protocols behind specific, domain-sensitive applications of computerized behavior.

I. Robinson (✉)
Neo Technology, Menlo Park, CA, USA
e-mail: iansrobinson@gmail.com

Domain application protocols specialize the interactions between the participants in a distributed application. This specialization is a good thing insofar as it helps support successful domain outcomes. Implemented unwisely, however, specialization inhibits a system's evolution and the serendipitous reuse of its components outside their original context. To overcome this problem, a RESTful API communicates specialization using several of the Web's more generalized mechanisms: namely media types, link relations and HTTP idioms. These artifacts help communicate a domain protocol without our having to import a specific process description into the client part of an application: the resulting domain application protocol is no more written on the surface of the API than a grammar is written on the surface of a sentence.

HTTP is *the* application protocol (Paul Prescod 2002), a domain-agnostic set of rules and conventions for accessing and manipulating resource representations in a uniform manner. Do we really need to introduce the concept of a domain application protocol when we already have the ubiquitous HTTP at our disposal? The answer, I believe, is: yes. Experience suggests that HTTP's domain agnosticism, while enormously beneficial in terms of standardization and interoperability, nonetheless leads to a shortfall in domain semantics. This shortfall must be remedied by every application in its own fashion, most often through prose documentation. HTTP doesn't tell us how to publish web content [the Atom Publication Protocol (Gregorio and de hOra 2007) remedies that], or how to manage cloud resources [The Sun Cloud API (2009) remedies that], or how to procure goods. To achieve a degree of specialization, both AtomPub and Sun's Cloud API apply specific web artifacts – HTTP idioms, media types, and, in the case of AtomPub, link relations – to achieve specific application goals. To retain the generalized benefits of HTTP's uniform interface, both require clients to bind to these web artifacts, rather than to the domain protocols themselves. In doing so, neither restricts a client from applying a system's resources in other contexts and for other purposes. This is the very same approach that I adopt here.

What Is a Domain Application Protocol?

To answer this question, consider the business process shown in Fig. 3.1.

Figure 3.1 illustrates the sequence of interactions that must take place for a customer to purchase some goods from a supplier.[1] The customer asks the supplier for a quote. On receiving a quote, the customer decides to order the goods for which they have been quoted. Once the supplier has confirmed the order, the customer pays for the goods, or cancels the order.

[1]This example simplifies the set of interactions encountered in a real-world application in order to highlight the key points in protocol design.

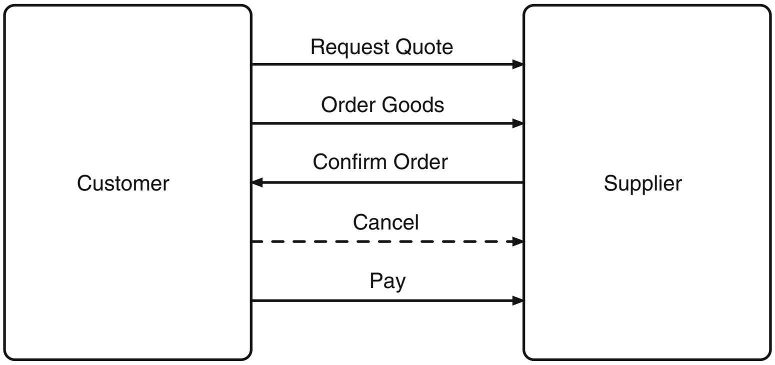


Fig. 3.1 A simple procurement process

Imagine that we have been charged with exposing this procurement process to third parties over the Web. A specific business need – the desire to allow customers to order and pay for goods – motivates a specific engineering task: that of exposing our quoting, order processing and payment functions in a way that allows customers to execute and complete our procurement process in a repeatable, well-understood manner. At the same time, however, we must remain mindful of the fact that other applications may wish to reuse parts of our system for entirely different purposes. Despite having been motivated by the specific business need behind this first project, we do not want to overly specialise our systems' interfaces; rather, we want to implement our APIs in ways that allow them to be composed into other applications and processes.

Fast forward to a time when we've built and deployed a solution to meet our business' procurement needs, and a client has just successfully completed an instance of our procurement process. In order to reach the successful conclusion of the process, the client had to initiate a series of legitimate interactions with whatever systems we'd chosen to expose over the network. The successful completion of the process implies the effective existence of a *domain application protocol*, a set of rules and conventions through which participants in a distributed system coordinate their interactions to achieve a useful, domain-specific application goal.

In the context of a RESTful web application, a domain application protocol is an abstraction of the media types, link relations, and HTTP idioms necessary to achieve a particular application goal. The design of a domain application protocol incorporates the deliberate discovery activities necessary to describe a RESTful API in terms of specific media types and links relations, plus a context-sensitive narrowing of HTTP.

Application

We call the actual occurrence of a set of interactions between participants in a distributed system an *application*. An application, in other words, is computing in action: computerized behavior directed towards achieving a particular client or end user goal. A distributed application is one in which multiple participants employ computing behavior to realize an application goal. By this definition, an application is not so much a *thing* as a *doing*; it is the very act of putting software to work to realize some benefit. Importantly, an application has duration – it unfolds in time.

Application State

Application state is a snapshot of the state of a distributed application at a particular point in time. Because an application has duration, its state changes over time. Once an application's goal has been achieved, the application can be considered to be in its final state. Prior to achieving this final state, the application passes through one or more intermediate states.

In the context of a conversation between participants in a distributed application, we can also think of application state as being the state of the conversation at a particular point in time. In this respect, application state guarantees the integrity of a sequence of requests. For example, if a client obtains an authenticated token at a certain point in a conversation, it can supply this token with all subsequent requests. Each request then contains sufficient application state information for the server to handle the request without recourse to a server-side session store.

Domain Application Protocol

A domain application protocol is the set of rules and conventions that guides and constrains the interactions between participants in a distributed application.[2] By adhering to a protocol, participants achieve a useful domain or business outcome. Revisiting our definition of application, we can say that an application is an *instance* of the execution of a protocol. In executing the protocol, the participants create an application, which in turn achieves an application goal.

To achieve an application goal in the context of a RESTful web application, a client progressively interacts with a community of resources. These resources can be hosted and governed by a single server, or they can be distributed across the network. Either way, every resource implements the same uniform interface, which in the case of a web application is HTTP.

[2]The term "domain application protocol" and the three-step design methodology described here were first proposed in Webber et al. (2010). We chose the term "domain application protocol" so as to align it both with the book's focus on automating business (domain) processes, and with our use of the terms "application" and "application state." A domain application protocol is more commonly referred to as a coordination protocol: see, for example, Alonso et al. (2004).

Application State in a RESTful Application

Having a server remember the state of each client conversation is expensive, particularly at web scale. To alleviate this burden, a RESTful web application delegates the responsibility for remembering the overall state of an application to the client or clients participating in that application.

As a host of application state, the client in a RESTful web application is responsible for the integrity of a sequence of actions. After each interaction the client is presented with one or more options to interact with additional resources. Servers encode these options in responses using links and forms – otherwise known as *hypermedia controls*.[3] The client decides which control to operate based on its understanding of the current state of the application.

Occasionally, a client may need to add some portion of application state information to its next request in order to provide sufficient application state context for the processing of that request. For example, if the client has received an authorization token in a previous response, it might add this token to all subsequent requests (by sending it in an `Authorization HTTP` request header), thereby conveying to the server the portion of application state information necessary to handle the request.

Design Steps

When automating multi-party business procedures in a RESTful web application, the following three-step process can help guide our design and implementation activities:

1. Model applications as application protocol state machines.
2. Implement them based on resources, resource life cycles and the server-governed rules that associate resources.
3. Document and execute them using media types, link relations, and HTTP idioms.

Step 1 is concerned with the design of an abstract domain application protocol. This step is accomplished without reference to any particular architecture or technology. Steps 2 and 3, on the other hand, focus on the choices particular to the design of a RESTful application, with Step 3 concentrating on the elaboration of a RESTful API.

In practice, the design and implementation of a RESTful web application will not always follow this three-step process. Step 1 in particular is often omitted. For applications whose protocols are relatively trivial, this is perfectly acceptable. Such is the case with simple data services: CRUD (Create, Read, Update and Delete) is a

[3]See Chap. 5, "Hypermedia Types," for a more thorough, and more nuanced, discussion of hypermedia control capabilities.

protocol, albeit a very simple one. Most CRUD-based data services are designed and implemented without reference to the underlying domain application protocol. This doesn't, however, mean that there isn't a protocol – only that we haven't modelled it explicitly. Every application is an instance of a protocol, no matter how simple or implicit.

Whereas Step 1 is optional, Steps 2 and 3 usually proceed iteratively and in parallel. We start by identifying a number of candidate resources, and then detail the HTTP interactions through which a client manipulates representations of these resources. In working through these interactions, we discover additional resources that help adapt the domain to the goals expressed in the protocol. In the worked scenario later in this chapter, for example, we discover some forms-based resources; these resources allow a client to request a quote, submit an order, and cancel an order.

Step 1

As part of the process of articulating a domain application protocol and understanding how it contributes to the successful achievement of an application goal, we create an application state machine representation of the state transitions to be coordinated by the protocol. It is important to point out here that this state machine representation of the protocol is neither an implementation artefact nor public documentation; it simply aids analysis. By explicitly modelling a protocol as a state machine, we gain a better understanding of the "value stream" of application state transitions through which value is released both to the customer and to the organisation(s) owning a process.

Figure. 3.2 shows the several different application state transitions that occur when we execute our procurement protocol. The application terminates when it is in either a *Paid* or a *Cancelled* state. Prior to that, the application passes through the *Quote Requested*, *Goods Ordered* and *Order Confirmed* states.

A procurement application passes through these several different states no matter how it is implemented. Each state refers to the state of the distributed application as a whole (the system), rather than to the state of an individual participant (customer or supplier) or entity (quote, order or payment).

Step 2

On the server side, a RESTful web application is based around resources and resource life cycles.

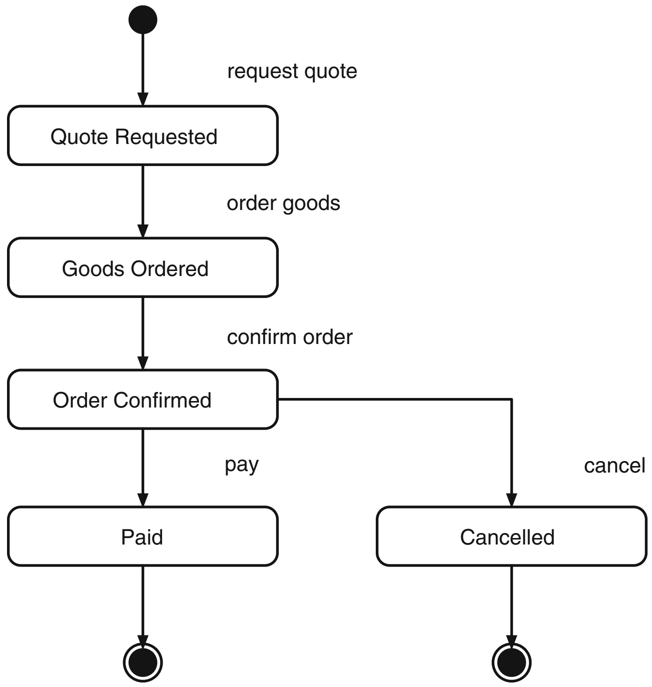


Fig. 3.2 Procurement protocol domain application state machine

Resources

Proponents of web-based systems define the resource abstraction in several different ways. In the most general definition, a resource is simply anything that can be identified by a URI (Berners-Lee et al. 2005). Such a definition lends itself to an inside-out, server-centric view, which sees resources as stateful "things" residing on the server. In contrast, the REST thesis (Fielding 2000) defines a resource as being a membership function, which groups a set of equivalent resource representations and identifiers. Membership of this set can vary over time. In a similar vein, (Booth 2006) sees a resource in terms of a set of state-dependent network functions that accept and return representations. Complementing these several viewpoints, I propose that resources be understood less in terms of what they *are*, and more in terms of what they *do*; resources *adapt* server-based capabilities so that hypermedia clients (i.e., clients that use HTTP's uniform interface to drive an application forwards) can use them.

A hypermedia client applies networked data in pursuit of its application goals. Consequently, a hypermedia system can be regarded as the partial application of networked data to a client or end user goal. Each response to a client request comprises a partial data structure; partial insofar as some of the data items represent links or forms that must be activated to retrieve or produce more data. Clients extend the structure by applying some of this data back to the network through the uniform interface. Applying the data – operating a link or form – only partially completes the structure; more often than not, it reveals yet more links and forms.

By emphasizing the resource's role in adapting server capabilities for consumption by network-oriented clients, we address one of the downsides of the server-centric perspective, which is the tendency to treat resources in terms of a relatively closed set of domain entities, coarsely manipulated through a small set of verbs. While suitable for simple CRUD-based data services, this entity-oriented attitude to building RESTful systems fails to address the needs of more sophisticated processes. The protocol perspective suggests that resources do not map directly to domain entities; rather, they serve to adapt the domain for its partial application through hypermedia and the uniform interface. Adapting a domain for consumption by a hypermedia client results in our identifying more resources than would normally be identified through a domain-entity-oriented approach. From the client's point of view, domain (i.e. business) behaviors emerge as a side effect of applying a relatively closed set of document-oriented verbs to this open set of resources.

Resource State

A resource has state, and this state, much like application state, can have its own lifecycle. But whereas application state lends integrity to a sequence of interactions with multiple resources, resource state is concerned solely with the state of an individual resource. This resource state is governed and maintained by the server hosting the resource. Attempts to manipulate a resource's state representations must conform to the business rules the server uses to govern the lifecycle of the resource. Such business rules are private to the server and should never be exposed to clients.

For most resources, a resource's state is simply a function of its data. For some resources, however, a resource's state is also partly a function of the state of other resources with which the resource is associated through some server-governed rules. For example, the state of an order is partly a function of the state of any payment with which that order has been associated by the server hosting the order. As with any other business rules governing the state of a resource, these rules remain hidden behind the RESTful interface.

Servers, then, are responsible for maintaining resource state, not application state. Application state remains significant, however. The overall distributed application still moves through several different application states. What's important is that the application state model (and the corresponding protocol) is nowhere reified on the server side. Changes to the state of the overall distributed application emerge as a *side effect* of the client manipulating the states of individual resources through their representations.

Through interacting with a community of resources, a client progressively realizes an application goal in accordance with an implicit domain application protocol. Some client interactions retrieve representations of resource state, others manipulate that state. Interactions that manipulate representations of resource state manifest an implicit domain application protocol such that resource state transitions occur in a legitimate sequence. It only makes sense to create a payment resource if one has first created an order with which the payment might be associated – and any good system design ought encourage this kind of behavior. How, then, do we encourage such behaviors in a RESTful web application?

Hypermedia

We coordinate a client's interactions with a community of resources by applying REST's hypermedia constraint (Fielding 2000) to the design of our resources and their representations. In this context, the hypermedia constraint is best summarized as, "hypermedia systems change application state."

A hypermedia system comprises a client, one or more server-governed resources, and some systemic behavior. This systemic behavior is initiated when a client makes a request of a resource – in a web application this will be a resource identified by a URI. The resource responds with a representation of its resource state. This representation includes one or more hypermedia controls – links and forms – which advertise legitimate interactions with other resources. The client processes the response and updates its understanding of the current state of the application. If it hasn't yet achieved its application goal, the client chooses the hypermedia control best suited to making forward progress, and operates that control. Operating the control triggers another request, and the cycle begins again.

When generating a response, the server that hosts and governs a resource uses the resource's state plus any application state information supplied by the client in the request to determine which controls to include in the response.

Step 3

A RESTful API is documented using media types, link relations and HTTP idioms.

Media Types

A media type value, such as `application/atom+xml`, is a key into a data format. While not all media types possess the capabilities necessary to implement a hypermedia system, those that do typically define one or more of the following:

- The format to be used for representing content.
- One or more schemas to which content must conform.
- Processing models for schema elements.
- Hypermedia control formats.
- Semantic annotations for hypermedia controls.

Fig. 3.3 A `<link>` element with semantic annotation

```
<link
  xmlns:rb="http://relations.restbucks.com/"
  rel="rb:order"
  type="application/restbucks+xml"
  href="http://restbucks.com/orders/9876"/>
```

The Atom Syndication Format (Nottingham and Sayre 2005), for example, includes all these elements. In terms of representation format, Atom is based on XML. With regard to schemas, the Atom specification includes two RELAX NG schemas: one for feeds, another for entries. To these it adds a processing model, which determines how content, foreign mark-up and extensions to the Atom vocabulary should be interpreted. In terms of its hypermedia capabilities, it identifies the `<atom:link>` element as a hypermedia control, and defines five link relation values (*alternate*, *related*, *self*, *enclosure*, and *via*) with which links can be annotated with semantic context. The Atom Syndication Format interpretative scheme is keyed off the `application/atom+xml` value in `Content-Type` request and response headers.

Link Relations

On the human web, we intuitively understand what links and forms mean based on the context in which they appear. Machines, on the other hand, cannot reliably infer such implicit semantics. In order to help machine clients decide which hypermedia control to activate in a received resource representation, we must provide some additional, explicit semantics. One of the most popular ways of adding semantic context to hypermedia controls is to annotate links with link relations.

Link relations describe the purpose of a link, the meaning of a target resource, or the relationship between a link's context and the target resource. By stating the purpose of a link, a link relation helps a client use the link according to its purpose. The semantic range of a link relation can vary from describing how the current link's context is related to another resource, to indicating that the target resource has particular attributes or behaviors.

HTML defined the `rel` attribute for annotating both anchor and link elements with link relations. This attribute convention was adopted by several other formats, including the Atom Syndication Format. Links that have been annotated with a link relation value are called typed links.

Figure. 3.3 shows a typed link taken from the example later in this chapter. The link has been typed with the link relation value *rb:order*. This value acts as a key into a semantic. In this instance, the associated semantic indicates that the linked or destination resource is an order.

Link relations come in one of two flavors: *registered* and *extension* (Nottingham 2010). Registered relations are registered with IANA's Link Relation Type registry (Link Relations 2011). Such well-defined link types take the form of simple string tokens. Examples of registered relation types include *self* and *payment*. Extension relations, on the other hand, are types that have not been registered with IANA. Such relations are often proprietary to an organisation or application. In order to disambiguate them from any similarly named relations elsewhere, they take the form of a URI. The link relation shown in Fig. 3.3 is an extension relation. It has been formatted as a compact URI (Birbeck and McCarron 2009); expanding the URI returns the absolute link relation value http://relations.restbucks.com/order.

Documenting a Protocol

A RESTful protocol is surfaced using an API composed of media types, link relations and HTTP idioms. Both the Atom Publication Protocol (AtomPub) (Gregorio and de hOra 2007) and Sun's Cloud API (The Sun Cloud API 2009) describe themselves in precisely these terms.

A protocol can draw on pre-existing media types and link relations, as well as invent its own. AtomPub is a good example of this compose-and-invent approach. AtomPub reuses the Atom media type, which is defined in the separate Atom Syndication Format specification; to this, it adds two new media types, `application/atomsvc+xml` and `application/atomcat+xml`, for representing service and category documents. To Atom's five link relations, AtomPub adds two more: *edit* and *edit-media*.

HTTP Idioms

A domain application protocol lends domain meaning to a distributed application's HTTP-based interactions. While all such interactions continue to adhere to the HTTP application protocol, their significance with respect to a client's application goal is determined by the domain application protocol. A domain application protocol constrains HTTP in the context of a particular application; from the client's perspective, this creates a temporally varying subset of HTTP idioms the client can use to manipulate representations of the resources participating in the protocol.

There are two approaches to communicating which HTTP idioms a client should use over the course of an application: upfront, and inline. With the upfront approach, we create a document describing the appropriate idioms. With the inline approach, we use HTTP headers and status codes, plus entity body control data, to communicate at runtime which idioms a client can use to manipulate resource representations.

The Atom Publication Protocol exemplifies the upfront approach. The AtomPub specification explicitly states that resources can be created with `POST`; that successfully creating a resource results in a response with a `201 Created` status code and `Location` header; that `PUT` and `DELETE` can be used to edit resources; and that all edits should be done in a conditional fashion (using an `If-Match` header with a previously supplied entity tag value).

The upfront approach determines which idioms are applicable to an application prior to any client beginning an application instance. In contrast, the inline approach effectively "programs" the client on the fly. The advantage of the inline approach is that it makes it easier to evolve and extend an application over time.

Here are some of the ways we can use HTTP headers, status codes and entity body control data to describe at runtime which HTTP idioms a client should use to manipulate resource representations:

- `Cache-Control` directives instruct intermediaries to cache content in accordance with HTTP's caching rules.
- Forms (HTML, XForms, etc.) program clients with control data (such as a URI, HTTP verb, and required `Content-Type` value), which the client can then use to encode and submit the form.
- `ETag` headers indicate to the client that subsequent requests for the same resource should use a conditional idiom: either a conditional `GET`, which uses an `If-None-Match` header with an entity tag value to instruct the server to return a full-blown response only if the resource addressed in the request *has* changed since the entity tag value was issued; or a conditional `PUT, POST` or `DELETE`, which uses an `If-None` header with an entity tag value to instruct the server to apply the request if *and only if* the resource addressed in the request *has not* changed since the entity tag value was issued.
- Some of the HTTP status codes determine the next HTTP idiom to be used; `303 See Other`, for example, instructs the client to issue a `GET` request for the resource specified in the accompanying `Location` header.
- `405 Method Not Allowed` tells the client that the verb in the request cannot be used; issuing an `OPTIONS` request for the same resource will return a `200 OK` response with an `Allow` header specifying which verbs can be used.

A RESTful Procurement Application

The remainder of this chapter comprises a narrative exposition of a set of HTTP interactions through which a client executes an instance of our procurement protocol. The example is set in Restbucks, a fictional coffee shop in a world of coffee-loving HTTP robots.[4] Starting from modest roots, Restbucks now has a number of retail outlets. Recently, it has decided to sell coffee beans direct to consumers.

In the course of this narrative, we'll see how the state of the procurement application changes as a result of the client accessing and manipulating representations of resource state. The narrative represents not so much a documented design as it does an act of deliberate discovery, as per the three-step methodology outlined earlier. We've already drawn the abstract protocol (Fig. 3.2) for Step 1: in the narrative that follows we identify a candidate set of resources, media types, link relations and

[4]Restbucks served as the basis for the examples in Webber et al. (2010).

```
Request:
GET /shop HTTP/1.1
Host: restbucks.com

Response:
HTTP/1.1 200 OK
Date: Mon, 26 Jul 2010 10:00:00 GMT
Cache-Control: public, max-age=86400
Content-Type: application/restbucks+xml
Content-Length: ...

<shop xmlns="http://schemas.restbucks.com/shop"
  xmlns:rb="http://relations.restbucks.com/">
  <link rel="rb:rfq"
    href="http://restbucks.com/request-for-quote"
    type="application/restbucks+xml"/>
</shop>
```

Fig. 3.4 Client starts the application

HTTP idioms (Steps 2 and 3, performed in parallel). Throughout, we make design decisions regarding resource boundaries, the connections between resources, HTTP headers, representation formats, and the placement of links and forms – all of which help drive out an API which is both specialized to the protocol *and* amenable to serendipitous reuse.

In the accompanying diagrams, arrow-headed arcs represent requests, while nodes represent responses. A response is shown either as a document containing typed links or a form, or as a status code requiring further action from the client. The round-cornered dashed boxes represent application states. These application states are not built into any of the server-side resources; rather, they have been superimposed onto the diagrams from the perspective of a third-party observer of the entire distributed application.

Start

Every application needs at least one entry point, located at a well-known URI, through which a client can initiate a sequence of interactions – and our procurement application is no different. To start the application, clients navigate to http://restbucks.com/shop, as shown in Fig. 3.4.

The response shown in Fig. 3.4 includes two headers of note: `Cache-Control` and `Content-Type`.

The `Cache-Control` header influences the behavior of any caching intermediaries – local caches, proxies, and reverse proxies – along the request–response path. Caching allows us to store copies of a representation closer to clients, thereby helping to conserve bandwidth, reduce latency, and minimize load on the origin server. In this instance, the header includes two directives: `public`, which

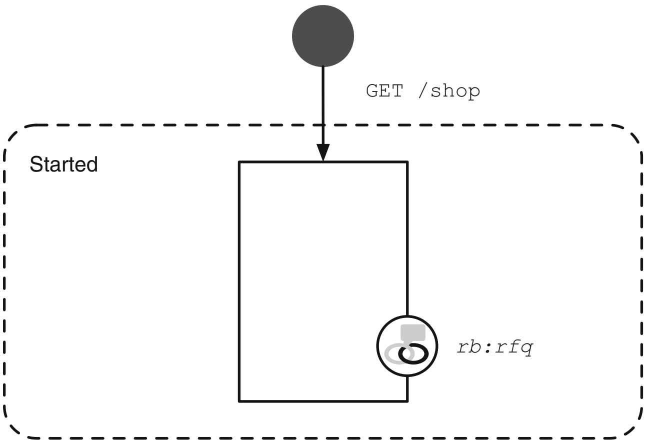


Fig. 3.5 Application begins in a *Started* state

makes the response cacheable by all intermediaries, both private and shared; and `max-age=86400`, which indicates that the response will remain fresh for up to one day after it was issued by the origin server. Together, these two directives ensure that the majority of requests for the procurement "homepage" are satisfied by the caching infrastructure, rather than by the origin server.

The response's `Content-Type` header has a media type value of `application/restbucks+xml`. This is a proprietary, but nonetheless reasonably generalized, format for representing quotes and orders; it is documented in more detail at the end of this chapter.

Below the response header block is the entity body, comprising an XML-formatted representation of the shop's homepage. This entry-point resource representation advertises the procurement application's capabilities. It currently contains a single `<link>` element. The link is typed *rb:rfq*, indicating that the resource at the other end of the link allows the client to request a quote.

An entry-point resource such as this is the ideal place to advertise new capabilities. If, for example, we were to evolve our application to include search functionality, we might advertise this new capability by adding a typed link (leading to a search form) to the shop's entry-point resource representation.

With this first client request, the overall distributed application enters the *Started* state, as shown in Fig. 3.5.

Request Quote

Having started the application, the client now processes the shop representation. The representation contains only one typed link, so to make forward progress, the client issues a request for the request-for-quote form, as shown in Fig. 3.6.

```
Request:
GET /request-for-quote HTTP/1.1
Host: restbucks.com

Response:
HTTP/1.1 200 OK
Date: Mon, 26 Jul 2010 10:00:05 GMT
Cache-Control: public, max-age=86400
Content-Type: application/restbucks+xml
Content-Length: ...

<model xmlns="http://www.w3.org/2002/xforms"
  schema="http://schemas.restbucks.com/shop.xsd">
  <instance/>
  <submission
    resource="http://restbucks.com/quotes"
    method="post"
    mediatype="application/restbucks+xml"/>
</model>
```

Fig. 3.6 Client gets a request-for-quote form

The response shown in Fig. 3.6 contains an XForms form (Boyer 2009). XForms is an XML vocabulary and data processing model for building web forms inside a host application. It is based on a model-view-controller architecture. The form shown in Fig. 3.6 uses an XForms `<model>` element to communicate control data to the client. The `<submission>` element's `resource`, `method` and `mediatype` attributes specify the URI, HTTP method and `Content-Type` header value to be used when submitting the form. The `<model>` element's schema attribute references an XML Schema instance to which the submitted content must conform. A client programmed with the correct media type library and appropriate HTTP and XForms processing capabilities can use this inline control data to compose and submit its next request.

Note that the representation format used here doesn't explicitly encode the fact that this form allows the client to submit a request for a quote – there's no `<request-for-quote>` element, for example. This is because throughout our procurement application we use a strategy of providing typed links to forms. The link relation associated with a typed link establishes the meaning of the linked resource. When dereferencing the link, the client retains this contextual knowledge (in this instance, the client understands that the linked resource will allow it to submit a request for a quote), and processes the received form accordingly. In doing so, the client navigates a steady state space. Following the link doesn't change the state of the overall distributed application; it does, however, enrich that state. By following a link to a form, the client discovers new opportunities – and appropriate idioms – for progressing the application further.

The client "fills out" the form – that is, it creates a request whose body conforms to the schema at http://schemas.restbucks.com/shop.xsd – and `POST`s it to the URI supplied in the control data, as shown in Fig. 3.7.

```
Request:
POST /quotes HTTP/1.1
Host: restbucks.com
Content-Type: application/restbucks+xml
Content-Length: ...

<shop xmlns="http://schemas.restbucks.com/shop">
  <items>
    <item>
      <description>Costa Rica Tarrazu</description>
      <amount>250g</amount>
    </item>
    <item>
      <description>Elephant Beans</description>
      <amount>250g</amount>
    </item>
  </items>
</shop>

Response:
HTTP/1.1 201 Created
Date: Mon, 26 Jul 2010 10:01:00 GMT
Location: http://restbucks.com/quotes/1234
Content-Type: application/restbucks+xml
Content-Length: ...

<shop xmlns="http://schemas.restbucks.com/shop"
  xmlns:rb="http://relations.restbucks.com/">
  <items>
    <item>
      <description>Costa Rica Tarrazu</description>
      <amount>250g</amount>
      <price currency="GBP">4.40</price>
    </item>
    <item>
      <description>Elephant Beans</description>
      <amount>250g</amount>
      <price currency="GBP">5.30</price>
    </item>
  </items>
  <link rel="self"
    href="http://restbucks.com/quotes/1234"
    type="application/restbucks+xml"/>
  <link rel="rb:order-form"
    href="http://restbucks.com/order-forms/1234"
    type="application/restbucks+xml"/>
</shop>
```

Fig. 3.7 Client submits a request for a quote

The resource at http://restbucks.com/quotes creates a quote based on the details supplied in the request. (Behind the RESTful interface, this resource contacts a quote engine to generate the quote.) The server returns a response with a `201 Created` status code, a `Location` header indicating the URI of the newly created quote, and an entity body containing a representation of the quote itself. This representation contains two typed links: a *self* link, which is the preferred URI for the quote, and an *rb:order-form* link. The *rb:order-form* link relation indicates that the resource at the other end of the link allows the client to submit an order based on the quote.

As an aside, it's worth noting that there's nothing special about the `POST` request that results from filling out the form shown in Fig. 3.6. In accordance with the XForms processing model, the `<model>` and `<submission>` scaffolding elements have been stripped away by the client. What ends up on the wire is simply the data representing a request for a quote. In other words, we could have added a typed link leading directly to the quotes resource to the application's homepage, and documented in our protocol specification that clients can POST a request for a quote to this linked resource. By using a typed link to a form, however, we avoid specifying specific HTTP idioms upfront. Instead, we put the control data in the form. The downside of using a typed link to a form is that it requires an additional request–response interaction – but given that the blank form is highly cacheable, the overhead of this additional request will be mitigated in many instances by the caching infrastructure.

With this `POST` request and response, the client sees that the overall distributed application's state has changed from *Started* to *Quote Requested*, as shown in Fig. 3.8.

Place Order

Assuming the quote is satisfactory, we can now observe what happens when the client wants to place an order. First, the client follows the quote's *rb:order-form* typed link, as shown in Fig. 3.9. The response contains another XForms form, similar to the one in Fig. 3.6. But whereas the form in Fig. 3.6 was empty, this one has been pre-populated by the server.

The response's `Content-Location` header indicates the source for this form data. The header value refers back to the quote issued earlier in the application. In other words, the entity encoded in the form is also accessible from another location: http://restbucks.com/quotes/1234. The result is that we have two resources, both of which share the same underlying domain data. The first adapts the domain so that a client can receive representations of a quote. The second – the pre-filled form – adapts the domain so that a hypermedia client can advance the procurement protocol by submitting an order based on a previously received quote.

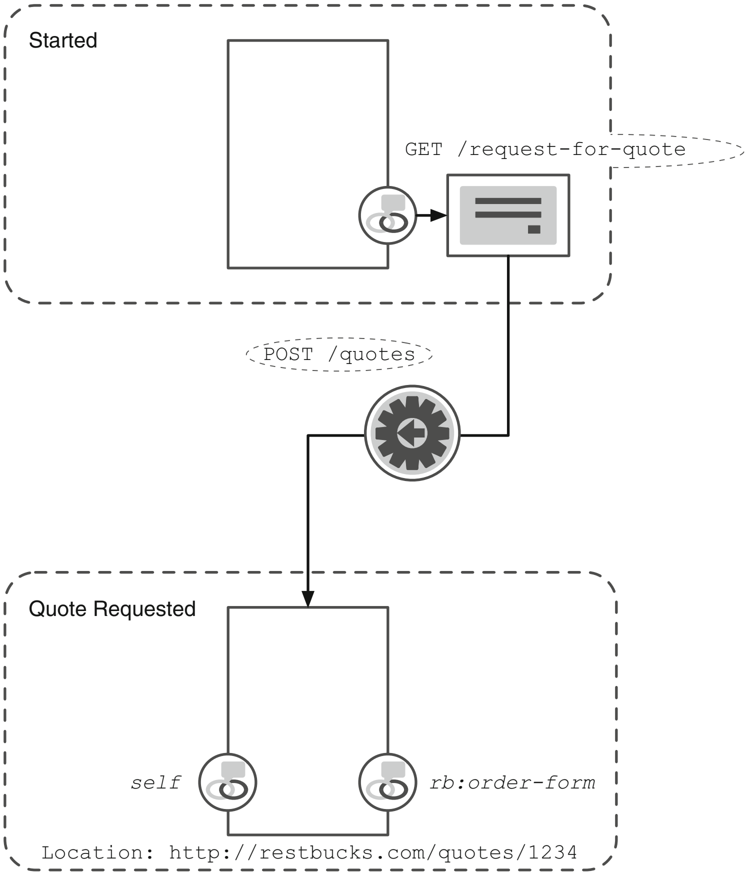


Fig. 3.8 Application state changes from *Started to Quote Requested*

Based on the quote data, the server responsible for this resource has generated a form that can then be POSTed to an order processor. The form contains all the information necessary to create an order, thereby eliminating any need for the order processor to look up the original quote. But this strategy, useful as it is in making the message self-sufficient, also raises an issue of message integrity, for the form's target need not be hosted on the same server – that is, the order processor may very well belong in an entirely different subsystem. Because we're passing around quote data, rather than a reference to a quote, a malicious client might be tempted to adjust the quote values prior to submitting the form, thereby earning itself a substantial discount. Given this possibility, how can we prevent clients from tampering with the message?

The solution we've adopted depends on the quoting and order processing subsystems having established a shared key. Prior to sending the response, the quotes resource generates a hash of the form data (the `<shop>` element and its

```
Request:
GET /order-forms/1234
Host: restbucks.com

Response:
HTTP/1.1 200 OK
Cache-Control: public
Date: Mon, 26 Jul 2010 10:01:05 GMT
Expires: Mon, 02 Aug 2010 10:01:00 GMT
Content-Type: application/restbucks+xml
Content-Length: ...
Content-Location: http://restbucks.com/quotes/1234

<model xmlns="http://www.w3.org/2002/xforms">
  <instance>
    <shop xmlns="http://schemas.restbucks.com/shop">
      <items>
        <item>
          <description>
            Costa Rica Tarrazu
          </description>
          <amount>250g</amount>
          <price currency="GBP">4.40</price></item>
        <item>
          <description>Elephant Beans</description>
          <amount>250g</amount>
          <price currency="GBP">5.30</price></item>
      </items>
      <link rel="self"
        href="http://restbucks.com/quotes/1234"
        type="application/restbucks+xml"/>
    </shop>
  </instance>
  <submission
    resource="http://restbucks.com/orders?c=99fe97e1
      ➥&s=k2awEpciJkd2X8rt3NmgDg8AyUo%3D"
    method="post"
    mediatype="application/restbucks+xml"/>
</model>
```

Fig. 3.9 Client gets the order form

children) and signs the hash using this shared secret. It then appends the generated value, together with its client ID, to the form URI, to make http://restbucks.com/orders?c=99fe97e1&s=k2awEpciJkd2X8rt3NmgDg8AyUo%3D. On receiving the POSTed form, the ordering subsystem is able to parse out the client ID and signed hash, recalculate its own version of the signed hash, and compare the recalculated value with the received value. [5]

[5]This is an example of a one-time URI. See Allamaraju (2010) for more details of generating one-time URIs.

Note that the design decisions we've made here trade message integrity for increased coupling. The quotes resource and the order processor are coupled through their sharing a secret to sign the hash, and through their sharing a URI template, `/orders?c={clientId}&s={signedHashValue}`, to generate the form URI. Moreover, if the shared secret leaks out, the tamper proofing mechanism will have been compromised.

There is one final thing to note about the response shown in Fig. 3.9. Restbucks has a business rule that says that a quote is valid for up to seven days after it has been issued. As we can see from the quote response in Fig. 3.7, the quote that was recently requested by the client was generated on Monday, 26 July 2010 at 10:01:00 GMT. The `Expires` header attached to the order form response indicates that the form representation can be cached, and will remain fresh, for exactly seven days from when the underlying quote was first issued.

To place its order, the client submits the form, as shown in Fig. 3.10. The order processor responds with `202 Accepted`, indicating that it has successfully received the request but has not yet finished processing it. Both the `Location` header and the typed link in the response body point to a resource that the client can later interrogate to discover the eventual result of processing the request.

The `202 Accepted` status code separates the action of accepting the request from the work necessary to fulfil it. In doing so, it coordinates the successful transfer of the request in the context of an asynchronous server-side task. To create an order in its initial state, a number of potentially slow operations must take place behind the RESTful interface. The order processor must contact a third-party payment provider and set up a transaction (to be completed later by the client); it must also contact the warehouse to determine stock availability. Both of these operations are relatively slow. Rather than have the client hang onto a connection waiting for a response describing the outcome of all this work, we've chosen simply to acknowledge successful delivery of the request while queuing the work itself for subsequent processing.

With this interaction, the client's view of the state of the overall distributed application changes from *Quote Requested* to *Goods Ordered*, as shown in Fig. 3.11.

Confirm Order

The client can now begin to poll the resource identified in the `Location` header of the response shown in Fig. 3.10. In polling, the client becomes responsible for the successful "delivery" of the outcome of its order request. (In contrast, pub/sub solutions depend on either the publisher or a piece of middleware to deliver notifications to subscribers successfully.) Figure. 3.12 shows the client's first attempt at polling the order at http://restbucks.com/orders/9876.

The server responds with `404 Not Found`, indicating that the order has not yet been created (the tasks necessary to create the order in its initial state have not completed). The client waits a couple of seconds, and then tries again, as shown in Fig. 3.13.

```
Request:
POST /orders?c=99fe97e1&s=k2awEpciJkd2X8rt3NmgDg8Ay
➥Uo%3D HTTP/1.1
Host: restbucks.com
Content-Type: application/restbucks+xml
Content-Length: ...

<shop xmlns="http://schemas.restbucks.com/shop">
  <items>
    <item>
      <description>Costa Rica Tarrazu</description>
      <amount>250g</amount>
      <price currency="GBP">4.40</price>
    </item>
    <item>
      <description>Elephant Beans</description>
      <amount>250g</amount>
      <price currency="GBP">5.30</price>
    </item>
  </items>
  <link rel="self"
    href="http://restbucks.com/quotes/1234"
    type="application/restbucks+xml"/>
</shop>

Response:
HTTP/1.1 202 Accepted
Cache-Control: no-store
Date: Mon, 26 Jul 2010 10:02:00 GMT
Location: http://restbucks.com/orders/9876
Content-Type: application/restbucks+xml
Content-Length: ...

<shop xmlns="http://schemas.restbucks.com/shop"
  xmlns:rb="http://relations.restbucks.com/">
  <status>Initializing</status>
  <link rel="rb:order"
    href="http://restbucks.com/orders/9876"
    type="application/restbucks+xml"/>
</shop>
```

Fig. 3.10 Client submits the order form

This time, all the server-side tasks necessary to create an order in its initial state have been completed, so the server responds with a representation of the newly created order. As the value of the order's `<status>` element indicates, the order is *Awaiting Payment*. This is resource state – and a particularly interesting kind of resource state at that, for the state of this order is not only a function of the data proper to the resource, it is also (partly) a function of the state of the payment with which the order was associated when it was created. While the payment is waiting to be completed by the client, this order is in the state of *Awaiting Payment*. The

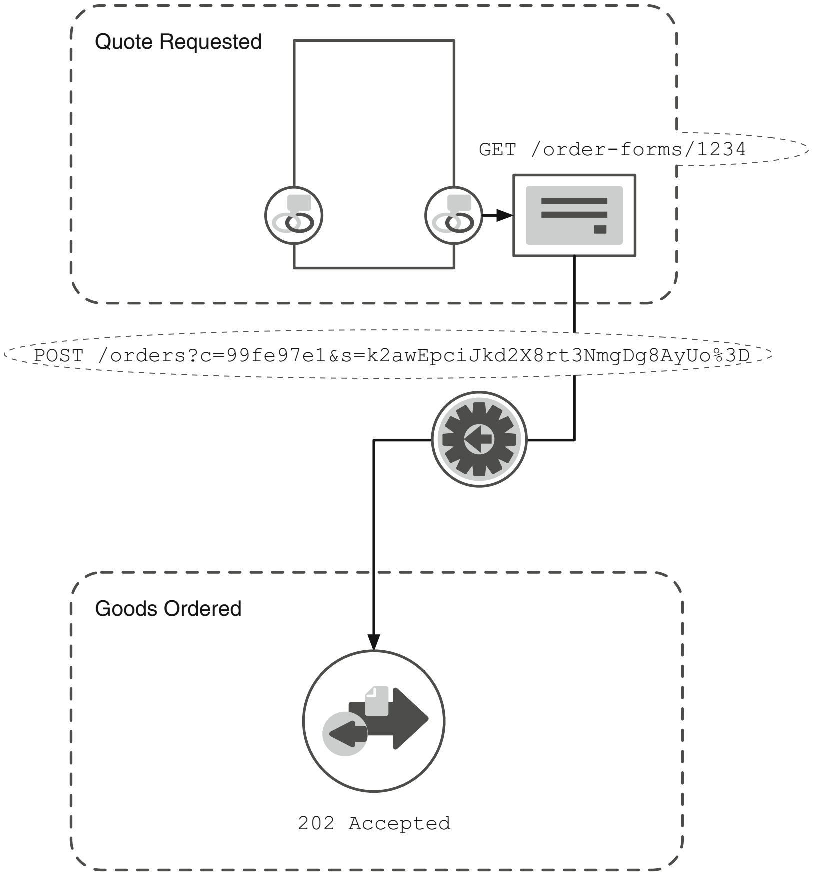


Fig. 3.11 Application state changes from *Quote Requested to Goods Ordered*

Fig. 3.12 The client polls the order resource

```
Request:
GET /orders/9876 HTTP/1.1
Host: restbucks.com

Response:
HTTP/1.1 404 Not Found
```

server responsible for the order resource can "watch" the payment resource in order to compute the state of the order.

The order's resource state, then, can change over time; moreover, it can change as a function of other resources changing state. This kind of situation requires us to make some tradeoffs between consistency and efficient use of network resources. The client here desires a view of the order consistent with the view held on the server; we, however, as designers of a networked application, want to use caching to conserve bandwidth, reduce latency, and save processing cycles.

```
Request:
GET /orders/9876 HTTP/1.1
Host: restbucks.com

Response:
HTTP/1.1 200 OK
Cache-Control: public, max-age=0
Date: Mon, 26 Jul 2010 10:05:00 GMT
ETag: "4d3e88c9"
Content-Type: application/restbucks+xml
Content-Length: ...

<shop xmlns="http://schemas.restbucks.com/shop"
  xmlns:rb="http://relations.restbucks.com/">
  <status>Awaiting Payment</status>
  <items>
    <item>
      <description>Costa Rica Tarrazu</description>
      <amount>250g</amount>
      <price currency="GBP">4.40</price>
    </item>
    <item>
      <description>Elephant Beans</description>
      <amount>250g</amount>
      <price currency="GBP">5.30</price>
    </item>
  </items>
  <link rel="self"
    href="http://restbucks.com/orders/9876"
    type="application/restbucks+xml"/>
  <link rel="rb:quote"
    href="http://restbucks.com/quotes/1234"
    type="application/restbucks+xml"/>
  <link rel="payment"
    href="https://example.org/payments/1010"
    type="application/xhtml+xml"/>
  <link rel="rb:cancellation"
    href="http://restbucks.com/orders/9876/cancel"
    type="application/restbucks+xml"/>
</shop>
```

Fig. 3.13 The client polls the order a second time

Fortunately, there is a way to provide both consistency and – to an extent – cacheability, using, as we have done here, `ETag` and `Cache-Control` headers.

The `ETag` header attached to the response in Fig. 3.13 contains an opaque string token – an entity tag value. An entity tag represents in digest form the state of a resource at the time the entity tag was generated. When the resource changes, the entity tag value changes. Clients and caches can use a previously supplied entity tag value to make efficient queries of the server governing the resource to which the entity tag belongs, as we'll see shortly.

Before we look at how a client or cache can use an entity tag value to maintain consistency in a reasonably network-efficient manner, let's examine the order's `Cache-Control` header. We've made the order resource representation cacheable using a cache-but-revalidate strategy, implemented using two `Cache-Control` directives. The first of these directives, `public`, makes the response cacheable by all caches; the second, `max-age=0`, indicates that the cached response must immediately be treated as stale.

This cache-but-revalidate strategy provides consistency, but at the expense of a small increase in network traffic. Anyone holding a copy of the order must revalidate with the origin server *with every request* using a conditional `GET`. Conditional `GET` requests look like normal `GET` requests, except they also include an `If-None-Match` header, which takes a previously received entity tag as a value. If the resource hasn't changed since the supplied entity tag was generated, the server responds `304 Not Modified`, thereby allowing the requestor to use its cached copy of the order. If the resource *has* changed since the supplied entity tag value was generated, the origin server replies with a full-blown response. This response travels all the way to the client, replacing any cached copies along the response path as it does so.

Returning to the entity body, we see that it contains four typed links: two with registered link relations (*self* and *payment*), and two with extension link relations (*rb:cancellation* and *rb:quote*):

- The *self* link indicates the preferred URI for the order.
- The *rb:quote* link points back to the quote used to created the order.
- The *rb:cancellation* link points to a resource that allows the client to cancel the order.
- The *payment* link refers to a resource that can be used to pay for the order.

With the transmission of the order response, the state of the overall distributed application has changed from *Goods Ordered* to *Order Confirmed*, as shown in Fig. 3.14.

Pay

Choosing now to pay for the order, the client `GET`s the *payment* typed link, as shown in Fig. 3.15. This request is made over a secure channel to a third-party payment provider.

The payment provider's response comprises an XHTML form representation of a payment waiting to be filled out with the client's payment details. The client fills out the form and `POST`s it back to itself. The outcome of this `POST` request depends on the current state of the payment. `POST`ing the client's payment details back to the payment resource for the first time changes the state of the payment from *Awaiting Payment* to *Paid*, and causes the payment to return a `200 OK` response, as shown in Fig. 3.16. Once is in the *Paid* state, however, the payment will no longer

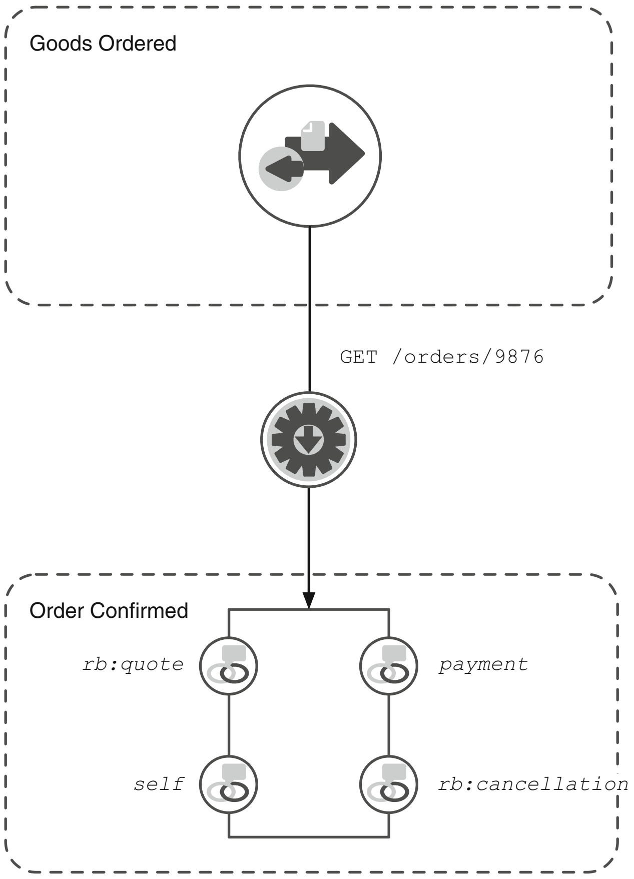


Fig. 3.14 Application state changes from *Goods Ordered to Order Confirmed*

accept `POST` requests; subsequent `POST` requests will cause the resource to return a `405 Method Not Allowed` response instead. In effect, the payment resource implements idempotent `POST`; that is, multiple `POST`s to the payment cause the transaction to be completed only once.

The response shown in Fig. 3.16 comprises another form. When the order processor set up the payment, it supplied the payment provider with a callback URI and confirmation ID. The payment provider uses these details to create a pre-filled payment confirmation form, which the client now submits, as shown in Fig. 3.17.

The resource to which the form data is `POST`ed validates the received confirmation ID and sets the state of the underlying order domain entity to *Paid*. It then redirects the client to the order resource with a `303 See Other` response. As shown in Fig. 3.18, the client makes a `GET` request of the URI supplied in the redirect's `Location` header.

```
Request:
GET https://example.org/payments/1010 HTTP/1.1

Response:
HTTP/1.1 200 OK
Cache-Control: no-store
Date: Mon, 26 Jul 2010 10:05:05 GMT
Content-Type: application/xhtml+xml
Content-Length: ...

<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Payment</title>
  </head>
  <body>
    <form method="post"
      action="https://example.org/payments/1010"
      enctype="application/x-www-form-urlencoded">
      <input type="text" name="card-type"/>
      <input type="text" name="name"/>
      <input type="text" name="card-number"/>
      <input type="text" name="security-code"/>
      <input type="submit" value="Submit"/>
    </form>
  </body>
</html>
```

Fig. 3.15 Client gets the payment form

When following the redirect to the order, the client adds the entity tag value it received the last time it requested the order to an `If-None-Match` request header, thereby making the request conditional. This conditional request requires the server to return a full-blown response only if the entity tag associated with the requested entity differs from the entity tag value supplied in the request. Because the order *has* changed since the client last requested it (its resource state has changed from *Awaiting Payment* to *Paid*, and therefore its entity tag value is different), the server returns a full response. This response includes a new entity tag value.

With this last series of interactions, the payment's state has changed to *Paid*, as has the order's. And with these two resource state changes, the client's view of the overall distributed application's state has changed from *Order Confirmed* to *Paid*, as shown in Fig. 3.19. The procurement application has reached a terminal state.

Cancel

Instead of paying for an order, a client may choose to cancel it. (In a real-world application there would likely be several points where the client could choose to cancel the order.) Following a link typed with *rb:cancellation* leads the client to a form, which the client then uses to `PUT` a reason for cancelling the order

```
Request:
POST https://example.org/payments/1010 HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: ...

card-type=Visa+Debit&name=MR+JOHN+SMITH&
➥card-number=4876512418675010&security-code=212

Response:
HTTP/1.1 200 OK
Cache-Control: no-store
Date: Mon, 26 Jul 2010 10:06:00 GMT
Content-Type: application/xhtml+xml
Content-Length: ...

<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Payment Confirmation</title>
  </head>
  <body>
    <form method="post"
      action="http://restbucks.com/payments/9876"
      enctype="application/x-www-form-urlencoded">
      <input type="hidden"
        name="confirmation-id">6a0806ca</input>
      <input type="continue" value="Continue"/>
    </form>
  </body>
</html>
```

Fig. 3.16 Client submits payment details

```
Request:
POST /payments/9876 HTTP/1.1
Host: restbucks.com
Content-Type: application/x-www-form-urlencoded
Content-Length: ...

confirmation-id=6a0806ca

Response:
HTTP/1.1 303 See Other
Date: Mon, 26 Jul 2010 10:06:02 GMT
Location: http://restbucks.com/orders/9876
```

Fig. 3.17 Client is redirected to the order

to a cancellation resource. This cancellation resource adapts the underlying order domain entity on behalf of clients wishing to cancel orders. Much as `POST`ting a payment confirmation modifies the underlying order and sets its state to *Paid* (as shown in Fig. 3.17), creating a new cancellation cancels the underlying order.

```
Request:
GET /orders/9876 HTTP/1.1
Host: restbucks.com
If-None-Match: "4d3e88c9"

Response:
HTTP/1.1 200 OK
Cache-Control: public, max-age=0
Date: Mon, 26 Jul 2010 10:06:05 GMT
ETag: "5612cfaa"
Content-Type: application/restbucks+xml
Content-Length: ...

<shop xmlns="http://schemas.restbucks.com/shop"
  xmlns:rb="http://relations.restbucks.com/">
  <status>Paid</status>
  <items>
    <item>
      <description>Costa Rica Tarrazu</description>
      <amount>250g</amount>
      <price currency="GBP">4.40</price>
    </item>
    <item>
      <description>Elephant Beans</description>
      <amount>250g</amount>
      <price currency="GBP">5.30</price>
    </item>
  </items>
  <link rel="self"
    href="http://restbucks.com/orders/9876"
    type="application/restbucks+xml"/>
  <link rel="rb:quote"
    href="http://restbucks.com/quotes/1234"
    type="application/restbucks+xml"/>
</shop>
```

Fig. 3.18 Order is now in a *Paid* state

Documenting the Procurement API

Having described a likely sequence of interactions through which a client can drive the procurement protocol forwards, together with the representation formats, processing models and link relation values necessary to realize these interactions, we're in a position to begin documenting the public face of our system. In large part, this documentation comprises descriptions of the media types and link relations we use throughout the application. It does *not* include any reference to the underlying protocol state machine. By coupling to our media types and link relations, clients allow themselves to be guided towards successfully completing the procurement protocol; at the same time, they are free to compose our resources and their interactions with those resources into entirely different applications.

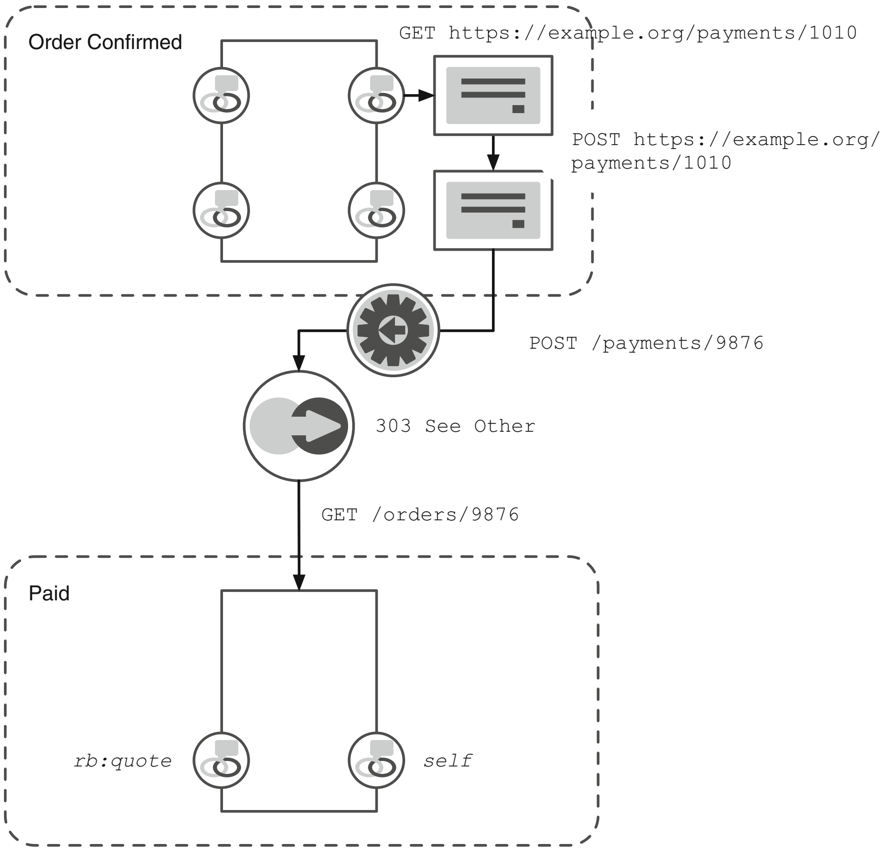


Fig. 3.19 Application state changes from *Order Confirmed to Paid*

The documentation we provide client developers indicates that our procurement application uses the `application/restbucks+xml` media type, together with a couple of registered link relations: *self* and *payment*. We also note that we use a third party payment provider whose protocol uses `application/xhtml+xml`.

The Restbucks Media Type

The documentation for the `application/restbucks+xml` media type says that:

- Responses will contain either a <shop> entity corresponding to the schema described at http://schemas.restbucks.com/shop.xsd, or an XForms `<model>`.

- We use `<link>` elements to represent links, and XForms `<model>` elements to represent forms and runtime control data.
- A `<shop>` may contain zero or more `<link>` elements, at most one `<items>` element containing zero or more child `<item>` elements, and at most one `<status>` element.
- We use five extension link relation values:

– http://relations.restbucks.com/quote – Indicates that the linked resource is a quote.
– http://relations.restbucks.com/order – Indicates that the linked resource is an order.
– http://relations.restbucks.com/cancellation – Indicates a resource where an order can be cancelled.
– http://relations.restbucks.com/rfq – Indicates a resource where a quote can be requested.
– http://relations.restbucks.com/order-form – Indicates a resource where orders can be submitted.

- User agents can automatically activate links typed with *rb:cancellation*, *rb:rfq* or *rb:order-form*. That is, these link relations indicate external resources that a client can prefetch to enrich its view of a steady state without changing the application's state.
- Clients wishing to use forms to further the application must understand and implement the XForms 1.1 Core Module.

With this documentation, client developers can develop media type libraries that parse and produce representations belonging to each media type, and which implement any processing models particular to those types; they can then compose these libraries into their client-side part of the application.

References

Subbu Allamaraju. *RESTful Web Services Cookbook*. O'Reilly, 2010.
Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer-Verlag, Berlin, Heidelberg, New York, 2004.
Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. 2005. http://www.ietf.org/rfc/rfc3986.
Mark Birbeck and Shane McCarron (eds). *CURIE Syntax 1.0*. 2009. http://www.w3.org/TR/curie/.
David Booth. *URIs and the Myth of Resource Identity*. 2006. http://dbooth.org/2006/identity/.
John M. Boyer (ed). *XForms 1.1*. 2009. http://www.w3.org/TR/xforms11/.
Roy Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.
Joe Gregorio and Bill de hOra (eds). *The Atom Publishing Protocol*. 2007. http://tools.ietf.org/html/rfc5023.
Link Relations. 2011. http://www.iana.org/assignments/link-relations
M. Nottingham. *Web Linking*. 2010. http://www.rfc-editor.org/rfc/rfc5988.txt.
M. Nottingham and R. Sayre (eds). *The Atom Syndication Format*. 2005. http://tools.ietf.org/html/rfc4287.

Paul Prescod. *Roots of the REST/SOAP Debate*, 2002 Extreme Markup Languages Conference, Montréal, Canada, Aug 2002.

The Sun Cloud API. 2009. http://kenai.com/projects/suncloudapis/pages/Home.

Jim Webber, Savas Parastatidis, and Ian Robinson. *REST in Practice: Hypermedia and Systems Architecture*. O'Reilly, 2010.

Chapter 4
Hypermedia Types

Mike Amundsen

The WWW is fundamentally a distributed hypermedia application.

– Richard Taylor

Hypermedia is defined by the presence of application control information embedded within, or as a layer above, the presentation of information.

– Roy T. Fielding

Abstract It is generally understood that, in the REST architectural style, "hypermedia is the engine of application state" (Fielding 2000). But what does that really mean? What is hypermedia? Can it be identified within a resource representation? How can hypermedia be the "engine of application state?"

Introduction

It is generally understood that, in the REST architectural style, "hypermedia is the engine of application state" (Fielding 2000). But what does that really mean? What is hypermedia? Can it be identified within a resource representation? How can hypermedia be the "engine of application state?"

In this chapter, a number of different notions of "hypermedia" along with a formal definition of "Hypermedia Type" will be presented. In addition, nine Hypermedia Factors (H-Factors) that can be found in resource representations are identified and examples of these factors are provided. Armed with these nine H-Factors, several registered media types are analyzed to determine the presence of these hypermedia elements and to quantify the hypermedia support native to these media types. Finally, a prototypical media type (*PHACTOR*) is defined and reviewed

M. Amundsen (✉)
Erlanger, KY 41018, USA
e-mail: mamund@yahoo.com

in order to show how H-Factors can be incorporated into a media type in order to produce a data format that can act as an engine of application state.

The Various Roles of Hypermedia

The history of hyper[*text*|*data*|*media*][1] is long and varied. Although a full treatment of the history of hypermedia is beyond the scope of this chapter, several aspects will be covered here. The first three are (1) hypermedia as read-only links, (2) hypermedia as GUI controls for local applications, and (3) hypermedia as state transition controls for components in a widely distributed network. In addition, the notion of hypermedia as an essential part of distributed network architecture as well as the use of MIME Media Types in HTTP is covered. Finally, a definition of "Hypermedia Type" will be presented.

Hypermedia as Links

The idea of hypermedia was given public voice by Vennevar Bush (1945) as a way to help researchers deal with what was perceived in the 1940s as an explosion of information. Bush described his idea for the "Memex" in a 1945 article, "As We May Think." In it he states, "The human mind ... operates by association. With one item in its grasp, it snaps instantly to the next that is suggested by the association of thoughts, in accordance with some intricate web of trails carried by the cells of the brain." (Bush 1945). He wanted to make it possible for information to be shared (using microfilm) and loaded into personal readers that could find links between subjects and allow the user to easily jump from one document to the next, following a single line of thought through a vast array of available content.

Decades later, in the 1974 self-published work, Computer Lib/Dream Machines (Nelson 1974), Theodor Nelson echoed Bush claiming "...writers do better if they don't have to write in sequence ... and readers to better if they don't have to read in sequence..." In this same work, Nelson coins the terms "hypertext" and "hypermedia" saying "By 'hypertext,' I mean non-sequential writing – text that branches and allows choices to the reader, best read at an interactive screen. As popularly conceived, this is a series of text chunks connected by links which offer the reader different pathways." (Nelson 1974).

In both these examples, hypermedia is thought of as a way to provide links between related materials and enable readers to move freely along these related paths. Hypermedia, in this case, is limited to a read-only experience meant for enriching reading, discovery, and comprehension of text.

[1]The words "hypertext", "hyperdata" and "hypermedia" all have seen active use; sometimes to mean different things. In this chapter, the word "hypermedia" indicates the general concept of links that provide 'jumps' or branches in text or any visual display. Therefore, throughout the rest of this chapter "hypermedia" will be used exclusively.

Hypermedia as GUI Controls

While the movement to enable improving the use-ability of text was underway, a second line of thought was also taking shape. That of using hypermedia as a way to control the location and retrieval of data for display to the user: hypermedia as a feature of graphical use interfaces.

As a radar station operator in the Philippines during World War II, Doug Engelbart happened upon Vannevar Bush's magazine article and was fascinated with the idea of the "Memex." Years later, Engelbart would publish "Augmenting Human Intellect: A Conceptual Framework" (Engelbart 1962) where he laid out his interpretation of Bush's vision.

"Most of the structuring forms ... stem from the simple capability of being able to establish arbitrary linkages between different substructures, and of directing the computer subsequently to display a set of linked substructures with any relative positioning we might designate among the different substructures." (Engelbart 1962).

By 1968, Engelbart had developed the NLS (oN-Line System) for sharing research information. His demonstration included not just a computer information system capable of supporting links between items, but the first "mouse" pointing device that could be used to actuate those links on screen. A video demonstration of this early hypertext system is still available for viewing (Engelbart 1968).

Later, in 1987, Jeffrey Conklin published "Hypertext: An Introduction and Survey" (Conklin 1987) which described hypertext as "...a computer-supported medium for information in which many interlinked documents are displayed with their links on a high-resolution computer screen." Conklin's work focuses on the role hypertext plays in graphical user interfaces (GUIs) and their influence on user interfaces in general. Conklin also compares editing environments for hypertext content.

Additional development of the personal computer through the 1980s and early 1990s introduced more display options and additional ways to express hypermedia links. The concept of hypermedia was expanded to include actuating interface controls such as selectors and buttons and spawned an emphasis on visual controls users can activate at any time. Hypermedia had become more than linking text, it was also a visual "affordance" to animate user displays.

Hypermedia as Application Controls

At the same time personal computer displays were providing more graphical controls, early versions of the World Wide Web appeared. In 1991, Tim Berners-Lee's WWW was available and, by 1993 the NSCA Mosaic Web Browser had become the popular graphical user interface for the WWW.

Along with the assumed graphical link elements that allowed WWW users to "jump" from one document to the next (or within documents), Web browsers had the ability to render images within the current document and the ability to send content to the server using link templates implemented as forms with input elements users could fill in and submit.

With the introduction of forms, hypermedia was no longer limited to static, read-only experiences. Documents could be created that allowed users to send data as well as find and retrieve it. HTML, the *de facto* document format for the WWW, allowed for the inclusion of application controls along with human-readable text.

Years later, in a slide presentation to ApacheCon 2005 Roy Fielding would describe this use of hypertext: "When I say hypertext, I mean the simultaneous presentation of information and controls such that the information becomes the affordance through which the user (or automaton) obtains choices and selects actions." (Fielding 2008).

Hypermedia as Architecture

The notion of hypermedia as more than text, more than data, but also application controls that allow users to make choices along the way is an important requirement for RESTful implementations over distributed networks. In addition to representing the state of the requested resource, Hypermedia documents contain the affordances for changing the state of the application. It is the hypermedia that makes state transitions possible. This alters the role of server responses from simple data replies to that of an essential part of the network architecture.

The idea that data itself (whether simple state information or hypermedia controls) can be part of the network architecture was articulated by Fielding (2000) as "...the nature, location, and movement of data elements within the system is often the single most significant determinant of system behavior." Even more directly, Fielding continues "The nature of the data elements within a network-based application architecture will often determine whether or not a given architectural style is appropriate." Finally, Fielding identifies hypermedia specifically as the "engine of application state."

MIME Types, HTTP, and Hypermedia Types

As mentioned in Fielding's 2001 dissertation (Fielding 2000), "HTTP inherited its syntax for describing representation metadata from the Multipurpose Internet Mail Extensions (MIME)." For this reason, RESTful implementations over HTTP are tied to using MIME Media Types for representing requests and responses. The official registry for MIME media types at the Internet Assigned Numbers Authority (IANA), contains hundreds of media type formats and descriptions. It would seem anyone setting out to create a RESTful implementation would have no trouble finding a wide range of suitable media types for handling their hypermedia resource representations.

However, a cursory review of solutions on the Web reveals that a relatively small number (not including binary representations for images, archive formats, etc.) of registered media types are consistently used as resource representation formats. Media types from that list that are widely supported are HTML (Raggett et al. 1999), Atom (Nottingham et al. 2005), XML (Bray et al. 2008), and JSON (Crockford 2006). Why is this the case?

The reason for favoring these few types is not merely historical priority. HTML has been around since the very start of the WWW and Atom earned Standards Track status in 2005. XML (first approved in 1998) and JSON (approved in 2006) are also often-used structured data formats. Apart from varying origin dates, these four media types have another defining characteristic worth noting. HTML and Atom include, as part of their format, well-defined link elements with clearly-associated protocol semantics. On the other hand, XML and JSON have no such defined native elements.

Having protocol semantics (e.g. HTTP GET, POST, etc.) defined within, and bound to elements of, the media type is an important distinction. Media types that share this trait are uniquely capable of enabling Fielding's "engine of application state." They are *hyper*media types. This discovery leads to a simple, but useful definition of "Hypermedia Type":

> *A Hypermedia Type is a media type that contains native hyper-linking elements that can be used to control application flow.*

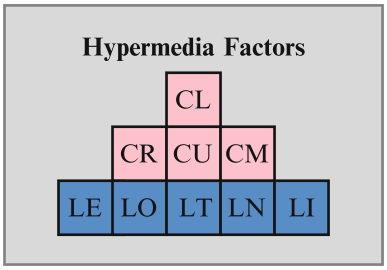
Summary

This section focused on various views of hypermedia itself; from read-only links to application controls. This last aspect of hypermedia – as a way to control applications through state transitions – has an important relation to architectural styles for distributed networks themselves. When responses are expressed as hypermedia documents, these responses are more than just data, they are also part of the network architecture.

Finally, a formal definition of the term "Hypermedia Type" was introduced.

Nine Hypermedia Factors

This section identifies nine factors native within a media type that can be used to support hypermedia behaviors. These nine "H-Factors" can be expressed in various ways, depending on the media type itself. However, no matter what actual elements or attributes are used to express these factors, the factors themselves remain essentially the same across all hypermedia types.



this figure will be printed in b/w

Each H-Factor identifies a clear hypermedia interaction between client and server. To this end, the H-Factors are divided into two distinct groups: "link" factors (`LO, LE, LT, LI, LN`) and "control data" factors (`CR, CU, CM, CL`). The five "link" factors denote specific linking interactions between parties: Outbound, Embedded, Templated, Idempotent, and Non-Idempotent, respectively. The remaining four "control data" factors provide support for customizing metadata details (e.g. HTTP Request Headers) of the hypermedia link interaction: Reads, Updates, Method, and Link Annotations.

Hypermedia factors		
Links	LO	Outbound Links
	LE	Embed Links
	LT	Templates Links
	LN	Non-Idempotent Links
	LI	Idempotent Links
Control data	CR	Read Controls
	CU	Update Controls
	CM	Method Controls
	CL	Link Annotation Controls

It should be noted that, to this author's knowledge there is no single registered media type that contains all nine of these factors. In fact, some media types contain none at all, some contain just one or two, etc.

Below is a list of the nine H-Factors, their descriptions and examples from well-known, registered media types.

Embedded Links: LE

The LE factor indicates to the client application that the accompanying URI should be de-referenced using the application-level protocol's read operation (e.g. `HTTP GET`) and the resulting response should be displayed within the current output window. In effect, this results in merging the current content display with that of the content at the "other end" of the resolved URI. This is sometimes called "transclusion."

A typical implementation of the LE factor is the `IMG` markup tag in HTML:

```
<img src="..." />
```

In the above example, the URI in the `src` attribute is used as the read target and the resulting response is rendered "inline" on the Web page.

In XML, the same LE factor can be expressed using the `x:include` element.

```
<x:include href="..." />
```

Outbound Links: LO

The LO factor indicates to the client application that the accompanying URI should be de-referenced using the application-level protocol's read operation and the resulting response should be treated as a complete display. Depending on additional control information, this may result in replacing the current display with the response or it may result in displaying an entirely new viewport/window for the response. This is also known as a "traversal" or "navigational" link.

An example of the LO factor in HTML is the `A` markup tag:

```
<a href="...">...</a>
```

In a common Web browser, activating this control would result in replacing the current contents of the viewport with the response. If the intent is to indicate to the client application to create a new viewport/window in which to render the response, the following HTML markup (or a similar variation) can be used:

```
<a href="..." target="_blank">...</a>
```

Templated Links: LT

The LT factor offers a way to indicate one or more parameters that can be supplied when executing a read operation. Like the LE and LO factors, LT factors are read-only. However, LT factors offer additional information in the message to instruct clients on accepting additional inputs and including those inputs as part of the request.

The LT element is, in effect, a link template. Below is an example LT factor expressed in HTML using the `FORM` markup tag:

```
<form method="get" action="http://www.example.org/">
  <input type="text" name="search" value="" />
  <input type="submit" />
</form>
```

HTML clients understand that this LT requires the client to perform URI construction based on the provided inputs. In the example above, if the user typed "hypermedia" into the first `input` element, the resulting constructed URI would look like this:

```
http://www.example.org/?search=hypermedia
```

The details on how link templates (LT) are expressed and the rules for constructing URIs depends on the documentation provided within the media type itself.

Templated links can also be expressed directly using tokens within the link itself. Below is an example of a templated link using specifications from the URI Template I-D (Gregorio et al. 2010):

```
<link href="http://www.example.org/?search={search}"/>
```

Non-Idempotent Links: LN

The LN factor offers a way to send data to the server using a non-idempotent "submit." This type of request is implemented in the HTTP protocol using the `POST` method. Like the LT factor, LN can offer the client a template that contains one or more elements that act as a hint for clients. These data elements can be used to construct a message body using rules defined within the media type documentation.

The HTML `FORM` element is an example of a non-idempotent (LN) factor:

```
<form method="post" action="http://example.org
/comments/">
  <textarea name="comment"></textarea>
  <input type="submit" />
</form>
```

In the above example, clients that understand and support the HTML media type can construct the following request and submit it to the server:

```
POST /comments/ HTTP/1.1
Host: example.org
Content-Type: application/x-www-form-urlencoded
Length: XX

comment=this+is+my+comment
```

It should be noted that the details of how clients compose valid payloads can vary between media types. The important point is that the media type identifies and defines support for non-idempotent writes.

Idempotent Links: LI

The LI factor provides a way for media types to define support for idempotent submits. These types of requests in the HTTP protocol are supported using the `PUT` and `DELETE` methods. While HTML does not have direct support for idempotent submits within markup (e.g. `FORM method="PUT"`), it is possible to execute idempotent submits within an HTML client using downloaded code-on-demand.

Below is an example idempotent link factor (LI) expressed using Javascript:

```
<script type="text/javascript">
  function delete(id)
  {
    var client = new XMLHttpRequest();
    client.open("DELETE", "http://example.org
    /comments/"+id);
  }
</script>
```

The Atom media type implements the LI factor using a `link` element with a relation attribute set to “edit” (`rel="edit"`):

```
<link rel="edit" href="http://example.org/edit/1"/>
```

Clients that understand the Atom specifications know that any link decorated in this way can be used sending idempotent requests (`HTTP PUT, HTTP DELETE`) to the server.

Read Controls: CR

One way in which media types can expose control information to clients is to support manipulation of control data for read operations. The HTTP protocol (Fielding et al. 1999) identifies a number of HTTP Headers for controlling read operations. One example is the `Accept-Language` header. Below is an example of XInclude (Marsh et al. 2006) markup that contains a custom `accept-language` attribute:

```
<x:include
  href="http://www.exmaple.org/newsfeed"
  accept-language="da, en-gb;q=0.8, en;q=0.7"
/>
```

Update Controls: CU

Support for control data during send/update operations (CR) is also possible. For example, in HTML, the `FORM` can be decorated with the `enctype` attribute.

The value for this attribute is used to populate the `Content-Type` header when sending the request to the server.

```
<form method="post"
  action="http://example.org/comments/"
  enctype="text/plain">
  <textarea name="comment"></textarea>
  <input type="submit" />
</form>
```

In the above example, clients that understand and support the HTML media type can construct the following request and submit it to the server:

```
POST /comments/ HTTP/1.1
Host: example.org
Content-Type: text/plain
Length: XX

this+is+my+comment
```

Method Controls: CM

Media types may also support the ability to change the control data for the protocol method used for the request. HTML exposes this CM factor with the `method` attribute of the `FORM` element.

In the first example below, the markup indicates a send operation (using the POST method). The second example uses the same markup with the exception that the GET method is indicated. This second example results in a read operation.

```
<form method="post" action="..." />
  <input name="keywords" type="text" value="" />
  <input type="submit" />
</form>

<form method="get" action="..." />
  <input name="keywords" type="text" value="" />
  <input type="submit" />
</form>
```

Link Controls: CL

In addition to the ability to directly modify control data for read and submit operations, media types can define CL factors which provide inline metadata for the links themselves. Link control data allows client applications to locate and understand the meaning of selected link elements with the document. These CL factors provide a way for servers to "decorate" links with additional metadata using an agreed-upon set of keywords.

For example, Atom (Nottingham et al. 2005) documentation identifies a list of registered Link Relation Values (IANA Protocol Registries 2011) that clients may encounter within responses. Clients can use these link relation values as explanatory remarks on the meaning and possible uses of the provided link. In the example below, the Atom `entry` element has a `link` child element with a link relation attribute set to "edit" (`rel="edit"`).

```
<entry xmlns="http://www.w3.org/2005/Atom">
  <title>Atom-Powered Robots Run Amok</title>
  <id>urn:uuid:1225c695-cfb8-4ebb-aaaa-
      80da344efa6a</id>
  <updated>2003-12-13T18:30:02Z</updated>
  <author><name>John Doe</name></author>
  <content>Some text.</content>
  <link rel="edit" href="http://example.org/edit/1"/>
</entry>
```

Clients that understand the Atom and AtomPub (Gregorio et al. 2007) specifications know (based on the documentation) that any link decorated in this way is the link to use when sending idempotent submits (`HTTP PUT, HTTP DELETE`) to the server.

Another example of using CL factors is HTML's use of the `rel="stylesheet"` directive (see below).

```
<link rel="stylesheet" href="..." />
```

In the above example, the client application (Web browser) will use the URI supplied in the `href` attribute as the source of style rendering directives for the markup in the HTML document.

Summary

This section identified nine Hypermedia Factors; one or more of which can be found in a media type document. The presence of these factors within the media type definition mark it as a hypermedia type and indicate support for various protocol-level hypermedia semantics. MIME media types that contain one or more of these factors promote RESTful implementations by making it possible to include application controls within the requests and responses. These hypermedia application controls are the elements of the message advance application flow.

Analyzing Media Types

Once a set of Hypermedia Factors have been defined, it is a simple matter to review any MIME media type and identify those H-Factors in the selected media type. By cataloging the H-Factors in a given type, architects and implementors can make assessments about the fitness of a particular media type for the intended implementation.

For example, an implementation that must support HTTP PUT and DELETE (H-Factor LI) should not rely solely on the HTML media type for resource representations since HTML has no native support for LI elements. Or, to use another example, if the proposed implementation requires support for templated links (LT), the Atom media type may not be the best selection for all use cases. However, by matching the hypermedia needs of the RESTful implementation to the H-Factors found in existing media types, a "best fit" of one or more media types can be identified for each use case.

Below are sample analyses of some registered MIME media types (URI List (Mealling et al. 1999), SVG (Dahlstrm et al. 2011), HTML (Raggett et al. 1999) and Atom (Nottingham et al. 2005).[2] In the interest of space, these media types are not exhaustively reviewed, but example elements are identified that meet the specifications of one or more of the H-Factors outlined previously in this chapter. This is done to give a general guide to the process of analyzing existing media types for the appearance of H-Factors as native elements.

Media Types Void of H-Factors

Some well-known media types are not covered here including XML (Bray et al. 2008) and JSON (Crockford 2006). These two (and similar ones) have been left out for an important reason: they contain no native H-Factors as part of their definition. In other words, these media types exhibit no defined elements capable of expressing any of the previously identified H-Factor links (LO, LE, LT, LI, LN) or control data (CR, CU, CM, CL).[3]

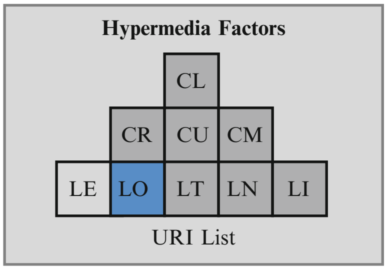
[2]It should be noted that these are not the only media types that warrant hypermedia analysis. They are also not selected here as excellent examples of Hypermedia Types, but merely as familiar media types worthy of review.

[3]While it is true that media types such as XML and JSON *allow* designers to define link and control elements *using* the basic elements of that media type, this does not qualify as providing native support for H-Factors.

URI List

A very simple example of a hypermedia type is the `text/uri-list` media type (Mealling et al. 1999). This media type consists of nothing more than a list of URIs:

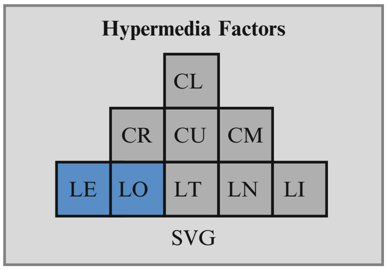
```
# urn:isbn:0-201-08372-8
http://www.huh.org/books/foo.html
http://www.huh.org/books/foo.pdf
ftp://ftp.foo.org/books/foo.txt
```



This media type is designed to convey a list of one or more URIs that can be resolved and/or processed by the recipient. For the sake of analysis, this media type provides support for the LO (Outbound Link) Hypermedia Factor. It might be argued that the URIs could be treated by recipients as LE (Embedded Links) (e.g. image links merged into an existing document), but most of the suggested uses in documentation point to de-referencing and processing each URI in turn rather than using the list to produce a single composite document.

SVG

Similar to the `text/uri-list` media type, the SVG media type (Dahlstrm et al. 2011) (`application/svg+xml`) exhibits support for a limited set of Hypermedia Factors. In this case, they are (1) the LO factor and (2) the LE factor.



The most common example of LO (Outbound Link) is the `A` element:

```
<a xlink:href="http://www.example.org">
  <ellipse cx="2.5" cy="1.5" rx="2" ry="1"
  fill="red" />
</a>
```

A common example of LO (embedded link) is the `image` element:

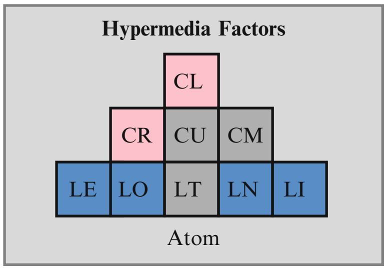
```
  <image x="200" y="200" width="100px" height="100px"
    xlink:href="myimage.png">
    <title>My image</title>
  </image>
```

While the SVG media type has a number of elements and attributes that support some form of URIs, all of these elements exhibit either the LO or LE H-Factors. Specifically, the SVG media type does not provide native support for LT (Templated Links), LI (Idempotent Link), or LN (Non-Idempotent Link) H-Factors.

Atom

The Atom media type profile is defined by two specifications Atom (Nottingham et al. 2005) and AtomPub (Gregorio et al. 2007). There are three registered media types associated withAtom: `application/atom+xml`, `application/atomcat+xml`, and `application/atomsvc+xml`. The Atom specification outlines the message format and elements for the `application/atom+xml` media type. The AtomPub specification covers the details for the remaining two media types as well as the read/write semantics for all three media types.

this figure will be printed in b/w



The primary hypermedia element in the Atom media type family is the `atom:link` element:

```
<link href="..." rel="..." hreflang="en" />
```

The `link` element show above supports LO, CR, and CL H-Factors. The Atom semantic model also supports LI and LN H-Factors by identifying markup elements within the response that have special significance.

For example, non-idempotent writes can be used to add new `entry` elements to the collection. Clients are instructed to locate the `atom:link` element associated with the root of the response which is marked with `rel="self"`. This element's `href` is the "Collection URI" and is the target URI for executing non-idempotent writes to the collection:

```
<feed xmlns="http://www.w3.org/2005/Atom">
  <link href="..." rel="self" />
  ...
</feed>
```

Idempotent writes (including updates and deletions) are indicated using the `rel="edit"` attribute on a `link` element that is the child of an `entry` element (see example below).

```
<entry>
  <link href="..." rel="edit" />
  ...
</entry>
```

There are a handful of other elements in the Atom media type family that support both LO and LE H-Factors including:

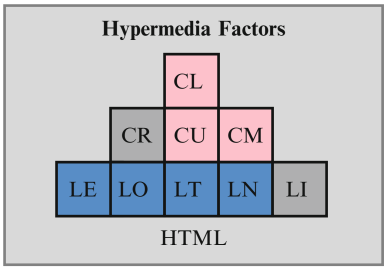
```
atom:collection (LO)
atom:content (LO,LE)
atom:generator (LO)
atom:icon (LE)
atom:logo (LE)
atom:uri (LO)
```

It should be noted that the Atom media type family uses documentation convention to communicate the details of a valid write payload (LI, LN) and does not include these details within the response message itself. Also, Atom has no native support for Templated Links (LT).

HTML

The HTML (Raggett et al. 1999) media type offers a wide range of support for H-Factors including: LO, LE, LT, LN and CU, CM, CL. The only H-Factors not supported in the HTML media type are LI (idempotent writes) and CR (control data for reads).

this figure will be printed in b/w



Simple outbound and embedded links are handled, for example, using the `A` and `IMG` tags respectively:

```
<a href="...">...</a>
<img src="..." />
```

HTML is the only media type covered in this chapter that supports Templated Links (LT). Templated links are similar to the LO H-Factor except that LT elements allow for URI variables to be defined by the sender and the values for these variables to be supplied by the client. In the HTML media type, this is accomplished using the `FORM` element with the `method="get"` attribute. The URI for the operation is found in the `action="..."` attribute. Below is an example:

```
<form method="get" action="http://example.org
/products" />
  <input type="text" name="color" value="" />
  <input type="text" name="size" value="" />
  <input type="submit" />
</form>
```

The HTML documentation instructs clients to use the inputs to amend the supplied URI before submitting the request to the server. Using the example above, and assuming input values, the constructed URI would look like the following:

```
http://example.org/products?color=red&size=large
```

Non-idempotent writes (i.e. the LN H-Factor) are supported using almost an identical markup as that seen for link templates. The only difference is the use of the `method="post"` attribute setting. For example, the same FORM element shown earlier can be used to write data to the server:

```
<form method="post" action="http://example.org/
  products"
  enctype="application/x-www-form-urlencoded" />
  <input type="text" name="color" value="" />
  <input type="text" name="size" value="" />
  <input type="submit" />
</form>
```

In the above case, HTML documentation instructs clients to use the inputs to construct a message body using (the default) `application/x-www-form-urlencoded` media type format. The resulting HTTP request sent to the server is:

```
POST /products HTTP/1.1
Host: example.org
Content-Type: application/x-www-form-urlencoded
Length: XX

color=red&size=large
```

It should be noted that, unlike the Atom media type which relies on documentation to tell clients how to compose a valid write message, the HTML media type allows servers to send clients write templates within the response and to use that template to compose a valid write request.

As seen in the two previous examples (LT and LN), HTML supports customizing control data protocol methods used for requests (CM) through the use of the `method` attribute of the `FORM` element and support for customizing control data for updates (CU) using the `enctype` attribute of the same element.

HTML also supports customizing control data for links (CL) using the `rel` attribute for the `link` tag. In the following example, the HTML client will use the response for the URI as style rules for adjusting the rendering of the content.

```
<link rel="stylesheet" href="... type="text/css" />
```

Summary

This section reviewed a representative sample of registered MIME media types and subjected them to a simple analysis in order to identify native elements within the media type that express one or more of the identified H-Factors. The analysis was cursory (in order to save time and space), but the reader should now have a good idea of how this can be undertaken for any registered (or proposed) media type design.

***PHACTOR*: A Prototypical Hypermedia Type**

Armed with the knowledge of the nine Hypermedia Factors and experience analyzing existing media types, it is possible to construct a prototypical media type that illustrates each of the H-Factors defined in this chapter.

The goal of this exercise is to create an "illustration" media type that can be used as a guide when setting out to analyze other media types or as an aide for those wishing to design their own Hypermedia Type. This prototypical media type contains all the identified H-Factors (LO, LE, LT, LN, LI and CR, CU, CM, CL).

What follows are the specifications for a media type called *Prototypical Hypermedia Application Controls for Text-Oriented Representations* or *PHACTOR*[4]. Like HTML, PHACTOR is a hypermedia type designed to support rendering and layout of text-based documents. For this reason, the reader will find many similarities between HTML and PHACTOR.

PHACTOR Layout Elements

PHACTOR is an XML-based media type used for representing simple text and a basic set of application controls. The main layout of a valid *PHACTOR* document is:

```
<document>
  <meta />
  <content />
</document>
```

The `meta` section can optionally hold one each of the following: `title, updated, author`.

```
<document>
  <meta>
    <title>H-Factor Sample</title>
    <updated>2010-06-15</updated>
    <author>MikeA</author>
  </meta>
  <content />
</document>
```

The `content` section can optionally hold one or more of the following elements (shown here in parent-child order): `section, para, text`. The `title` element may also appear as the first child of a `section` element. Also, the `eol` ('end-of-line') element can be used to create line breaks within a block of text.

```
<document>
  <meta>
    <title>H-Factor Sample</title>
    <updated>2010-06-15</updated>
    <author>MikeA</author>
  </meta>
```

[4]At the time of this writing, the *PHACTOR* media type has been submitted to the IANA Media Type registry with the `application/vnd.phactor+xml` MIME type identifier.

```
    <content>
      <section>
        <title>An implementation</title>
        <para>
          <text>
             This is a trivial hypermedia type
             implementation.<eol/>
             This is a new line in the document.
          </text>
        </para>
      </section>
    </content>
  </document>
```

PHACTOR Link Elements

In addition to simple text and layout elements, the *PHACTOR* media type supports the following link elements: `LO, LE, LT, LN, LI`. These link elements can appear as child elements of the following elements: `content, section, para`. Also, the `LT, LN, LI` elements may have one or more optional `data` child elements. The `data` can be used to create link templates (`LT`) and to populate `LN` and `LI` representations to send to the server.

Below are examples of each of the link elements. Note that both the `LT` and `LN` elements use `data` child elements to define templates.

```
<LO href="http://example.org" label="example.org" />

<LE href="http://example.org/images/photo.jpg"
label="Photo" />

<LT href="http://example.org/search" >
  <data name="keyword" label="Search" />
</LT>

<LN href="http://example.org/comments/" label="Add
Your Comment">
  <data name="nickname" label="Nickname" />
  <data name="comment" label="Comment" />
</LN>

<LI CM="delete" href="http://example.org/comments
/123" label="Delete Comment"/>
```

PHACTOR Control Data Elements

The *PHACTOR* media type supports optional control data for linking elements `CR, CU, CM, CL`. The `LE` element supports the `CR="[acceptLanguage]"` attribute to allow customizing the `Accept-Language` header of the request. The `LI` element supports the `CM="[protocolMethod]"` and `CU="[contentType"]` attributes to allow customizing the request with the protocol method and content type string respectively.

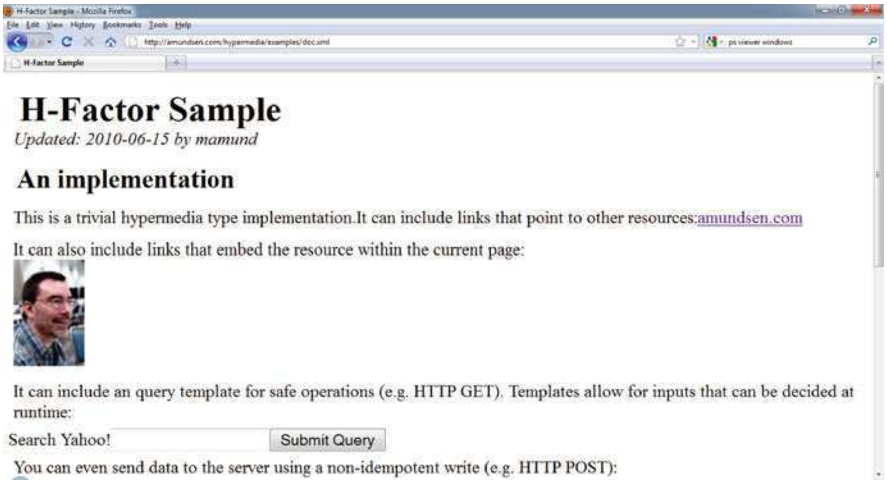
All five link types (`LO, LE, LT, LI, LN`) support the use of the `CL = "[linkRelationValue]"` attribute in order to decorate hypermedia links with additional metadata. Valid values for this attribute can be any registered link relation value or any fully-qualified unique URI [per RFC5988 (Nottingham 2010)]. It is up to the client application to determine the importance and meaning of this value.

A Complete *PHACTOR* Document

Below is a complete sample *PHACTOR* document along with a screen-shot sample rendering of the same document in a Web browser.

```
<document>
  <meta>
    <title>H-Factor Sample</title>
    <updated>2010-06-15</updated>
    <author>MikeA</author>
  </meta>
  <content>
    <section>
      <title>An implementation</title>
      <para>
        <text>This is a trivial hypermedia type
        implementation.</text>
        <text>It can include links that point to other
        resources:</text>
        <LO CL="document" href="http://amundsen.com"
        label="amundsen.com" />
      </para>
      <para>
        <text>
          It can also include links that embed the
          resource
          within the current page:
        </text>
        <eol />
        <LE CL="document" href="http://amundsen.com/
        images/mca.jpg" label="mamund" />
      </para>
      <para>
        <text>
            It can include a query template read
            operations (e.g. HTTP GET).
            Templates allow for inputs that can be
            decided at runtime:
        </text>
      </para>
      <LT CL="search" href="http://search.yahoo.com
      /search" >
        <data name="p" label="Search Yahoo!" />
      </LT>
    </section>
  </content>
</document>
```

this figure will be printed in b/w


H-Factor Sample - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://amundsen.com/hypermedia/examples/doc.xml
ps viewer windows
H-Factor Sample
Submit Query

H-Factor Sample

Updated: 2010-06-15 by mamund

An implementation

This is a trivial hypermedia type implementation.It can include links that point to other resources:amundsen.com

It can also include links that embed the resource within the current page:

It can include an query template for safe operations (e.g. HTTP GET). Templates allow for inputs that can be decided at runtime:

Search Yahoo!

You can even send data to the server using a non-idempotent write (e.g. HTTP POST):

Rendering PHACTOR Documents

Since the *PHACTOR* media type is based on XML, it is relatively easy to load, parse, and render using modern Web browsers. All modern Web browsers support client-side XSLT transformations. For the *PHACTOR* media type, each response can be accompanied by an XSLT stylesheet directive like the one below:

```
  <?xml-stylesheet type="text/xsl" href="phactor.xsl"?>
```

This directive can be used by the browser client to transform the XML response into valid XHTML that can be rendered by the client application. Also, since *PHACTOR* has support for Idempotent Links (LI), a full-featured Web browser implementation requires use of the `XMLHttpRequest` API (van Kesteren 2010).

Below is a partial listing of the transformation document:

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">

<xsl:output method="html"
  media-type="text/html"
  doctype-public="-//W3C//DTD XHTML 1.0 Strict//EN"
  doctype-system="DTD/xhtml1-strict.dtd"
  cdata-section-elements="script style"
  indent="yes"
  encoding="ISO-8859-1"/>

  <xsl:template match="/">
    <html>
      <head>
        <xsl:apply-templates select="//meta" mode=
        "head" />
        <link href="doc.css" rel="stylesheet" type="
        text/css" />
      </head>
      <body>
        <xsl:apply-templates />
      </body>
    </html>
  </xsl:template>

  <!-- head items -->
  <xsl:template match="meta" mode="head">
    <title><xsl:value-of select="title" /></title>
    <meta name="updated" content="{updated}" />
    <meta name="author" content="{author}" />

  </xsl:template>

  ...
```

Summary

In this chapter the varying role of hypermedia was reviewed including usage to express read-only links, act as GUI controls, and as a way to provide support for state transitions between distributed components. Hypermedia was also viewed from the perspective of network architecture itself.

Special attention was given to MIME Media Types and their use in HTTP in order to carry hypermedia information. It was observed that only a subset of MIME media types exhibit native hypermedia elements and these media types were used as the basis for a formal definition of "Hypermedia Type."

Nine native Hypermedia Factors (H-Factors) were introduced to show how media types can express application controls for various state transition activities in a RESTful implementation. In addition, several well-known registered media types were analyzed for the presence of H-Factors in order to quantify support for hypermedia in each media type.

Finally, a prototypical hypermedia type (*PHACTOR*) was presented to illustrate how media types can be designed using H-Factors. A sample page was presented and a sample *PHACTOR* user agent implementation based on a modern Web browser's support for XSLT and XMLHttpRequest was discussed.

References

Bray, Tim, Ed. et al., *Extensible Markup Language (XML) 1.0 (Fifth Edition)*, http://www.w3.org/TR/xml/ (2008)

Bush, Vannevar, *As We May Think*, Atlanic Magazine, July 1945

Conklin, Jeff, *Hypertext: An Introduction and Survey* in IEEE Computer, 20(9), 17–41, September 1987

Crockford, Douglas, *The application/json Media Type for JavaScript Object Notation (JSON)*, http://tools.ietf.org/html/rfc4627 (2006)

Dahlstrm, Erik, et al., *Scalable Vector Graphics (SVG) 1.1 (Second Edition)*, http://www.w3.org/TR/SVG/ (2011)

Engelbart, Douglas, *Augmenting Human Intellect: A Conceptual Framework*, October 1962

Engelbart, Douglas, *The Demo*, http://sloan.stanford.edu/mousesite/1968Demo.html (1968)

Fielding, Roy Thomas, *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000

Fielding, Roy Thomas, *REST APIs must be Hypertext-driven*, http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-drivencomment-718 (2008)

Fielding, Roy Thomas, Ed. et al., *Hypertext Transfer Protocol – HTTP/1.1*, http://tools.ietf.org/html/rfc2616 (1999)

Gregorio, J., Ed. et al., *URI Template*, http://tools.ietf.org/html/draft-gregorio-uritemplate-04 (2010)

Gregorio, J., Ed. et al., *The Atom Publishing Protocol*, http://tools.ietf.org/html/rfc5023 (2007)

IANA Protocol Registries, *Link Relations*, http://www.iana.org/assignments/link-relations/ (2011)

van Kesteren, Anne, *XMLHttpRequest*, http://www.w3.org/TR/XMLHttpRequest/ (2010)

Marsh, Jonathan, et al., *XML Inclusions (XInclude) Version 1.0 (Second Edition)*, http://www.w3.org/TR/xinclude/ (2006)

Mealling, M. et al., *URI Resolution Services Necessary for URN Resolution*, http://tools.ietf.org/html/rfc2483 (1999)
Nelson, Theodor H., *Literary Machines*. Swarthmore, Pa.: Self-published (1974)
Nottingham, M., *Web Linking*, http://tools.ietf.org/html/rfc5988 (2010)
Nottingham, M., Ed. et al., *The Atom Syndication Format*, http://tools.ietf.org/html/rfc4287 (2005)
Raggett, Dave, Ed. et al., *HTML 4.01 Specification*, http://www.w3.org/TR/html401/ (1999)

Chapter 5
Beyond CRUD

Irum Rauf and Ivan Porres

Abstract REST web services offer interfaces to create, retrieve, update and delete information from a database (also called CRUD interfaces). However, REST web services can also be used to create rich services that offer more than simple CRUD operations and still follow the REST architectural style. In such a case it is important to create and publish behavioral service interfaces that developers can understand in order to use the service correctly. In this chapter we explain how to use models to design rich REST services. We use UML class diagrams and protocol state machines to model the structural and behavioral features of rich services. The design models are then implemented in Django Web Framework. We also show how to use the behavioral interfaces to implement a service monitor.

Introduction

The interface of a web service advertises the operations that can be invoked on it. A web service developer looking for a particular service finds the service over the web and integrates it with other services by invoking the advertised operations and providing it the required parameters.

Many RESTful web services present simple interfaces to create, retrieve, update and delete information from a database (also called CRUD interfaces). However, REST is not limited to simple CRUD applications. It is possible to create web services exhibiting a rich application state that still follow the REST architectural style, e.g., flight and hotel reservation systems, stock trading services etc. In such cases, it is important to create and publish behavioral service interfaces so other developers can understand how to use a service correctly. A behavioral interface

I. Rauf (✉)
Department of Information Technologies ICT, Abo Akademi University,
Joukahainengatan 3-5 A, FI-20520 ABO, Finland
e-mail: irauf@abo.fi

of a web service provides information about the order of invocation and about any special conditions under which interface methods can be invoked and their expected effect.

A REST interface should offer features of *addressability*, *connectedness*, *uniform interface* and *statelessness*. In order to provide these interface features for beyond CRUD REST applications along with behavioral interface specifications, we present a design methodology that caters to the REST design philosophy earlier in the development cycle (Porres and Rauf 2011). The design approach addresses modeling of REST features using UML (Unified Modeling Language) (OMG UML 2009), thus creating web services that are RESTful by construction. In this chapter, we overview the design methodology presented in Porres and Rauf (2011) and then detail how the design approach is implemented in Django Web Framework. The service monitor implemented in Django Web Framework checks the correctness of a service with regard to its design.

Modeling the RESTful way

Models represent a system in graphical notations that are easier to understand and communicate between system developers and with other stake-holders of the system. We use UML to model the structural and behavioral features of REST web service. UML is a standard modeling notation and is well-accepted by industry. It provides representation of the system in an abstract manner from different perspectives and also serves as part of the specification document (Mens and Gorp 2005).

The objective of this modeling activity is to represent a REST web service with UML models that provide features of a REST interface, i.e., *addressability*, *connectedness*, *uniform interface* and *statelessness*. Using these design models, we can create a web service that will exhibit REST features thus making it RESTful by construction.

The starting point of the modeling activity is an informal web service specification in natural language. This specification is used to model structural features as a conceptual resource model and behavioral features as a behavioral model of the web service. Both the models are built in parallel and refined iteratively.

REST web services expose their functionality through resources. We model these resources in our conceptual resource model. The conceptual resource model is represented by a UML class diagram and tackles the *addressability* and *connectivity* requirements of a REST interface. The behavioral specifications of an interface are represented with UML Protocol state machine. A protocol state machine contains a number of states with state invariants and transitions. Each transition is triggered by a method. In a RESTful interface, resources do not have different access methods, instead the standard HTTP methods are used. Our approach uses four HTTP methods, i.e., GET, PUT, POST, and DELETE, for retrieving and updating data in a resource. The behavioral model tackles with the *uniform interface* and *statelessness* features of REST style.

In the next two sections, we show how these models are developed. We use as example an imaginary hotel room booking (HRB) service. The service allows a client to book a room, pay for the reservation, and cancel it. It is a simplified pedagogical example, but it shows how to design a REST interface for a service with a complex application state.

Conceptual Resource Model

A RESTful web service is data-centric and exposes its functionality through resources. Each resource has a representation in the form of data attributes. These resources form part of the static structure of the web service. We represent this static structure as a conceptual resource model using UML class diagram. A UML class diagram represents classes and associations between them. An association defines a relationship between two classes by which one class knows about the other class (OMG UML 2009).

As a starting step we analyze the natural language specifications of the service and identify the resources. Any important information in a service interface is exposed as a resource. Each resource is shown as a class in the class diagram. Identifying resources can be an iterative process and as we analyze and design the behavioral model of a web service, we can add or remove the resources in its conceptual resource model. As a general practice, the number of resources can be increased to reduce the complexity of a service interface. Every piece of information that needs to be retrieved or manipulated by the users of the service is modeled as a resource.

Figure 5.1 shows the conceptual resource model of the HRB RESTful service.We have broken our HRB service into six (non-collection) resources, i.e, (*booking, room, payment, pwaiting, pconfirmation and cancel*). A user interested in retrieving certain information can invoke a GET method on that resource and get representation of resource as a response. For example, if a user is interested in knowing whether a booking is canceled or a certain payment is confirmed, she would invoke GET method on *cancel* or *pconfirmation* resource, respectively.

A resource can also be a collection resource that contains a group of other resources. A collection resource is identified from the specifications and stereotyped as <<*collection*>> in the conceptual model. In Fig. 5.1, *bookings* and *rooms* represent collection resources with the stereotype *collection* and are linked to child resources, *booking* and *room*, respectively. A collection resource has a cardinality of more than 1 on the association end of a child resource. A GET method on a collection resource returns a list of all the child resources it contains. For example, a GET method on *bookings* will give a list of all the *booking* resources that it contains.

The attributes that form representation of a resource are represented as attributes of a class. These class attributes would appear in the resource representation, i.e. an XML document or a JSON serialized object.

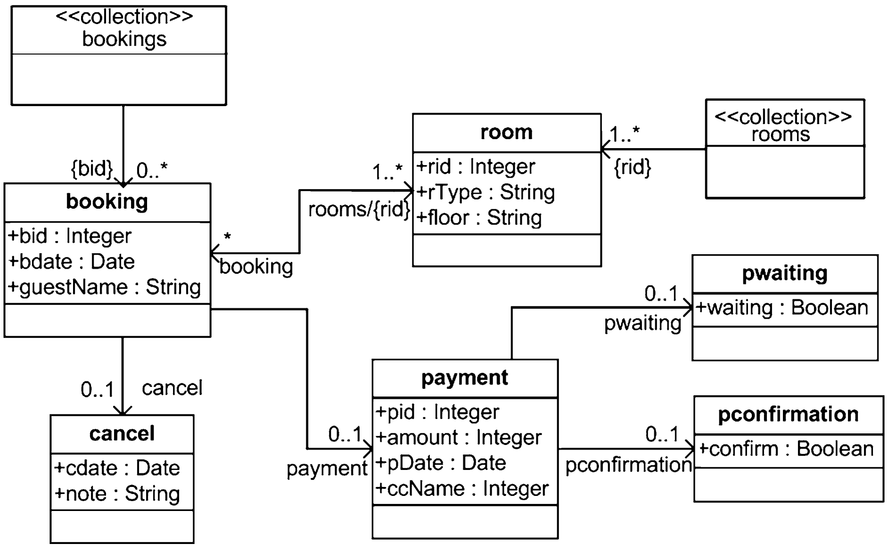


Fig. 5.1 Conceptual model for HRB RESTful web service

Figure 5.1 shows representation of resources in the HRB service. For example, *room* resource contains three attributes i.e. *rid, rType and floor*. Room ID(*rid*) and floor(*floor*) are integer values and room type(*rType*) is a string value. Attributes are modeled as a public attribute as the representation of a resource is available for manipulation.

Classes are connected via associations and each association is marked with role names on association ends. These associations show connection between resources and their multiplicity shows number of resources that can be related to the resource on the other end of the resource. These associations provide addressability and connectivity features to web service interface as explained in the next section.

Addressability and Connectedness

The associations between classes in the conceptual model provide information on the connection between the resources. The association direction shows the navigation direction and the role names on the association ends show the relative navigation path. Collection resources can be used as the starting point of the navigation paths to address each resource. Starting from a collection resource, we can access other resources by navigating the successive associations. For example, in Fig. 5.1, payment resource of a particular booking with id {*bid*} is retrieved by visiting the path */bookings/*{*bid*}*/payment/*. Paths visiting the same association more than once are not valid. In our example, the valid paths are listed below.

```
    /bookings/{bid}/
    /bookings/{bid}/cancel/
    /bookings/{bid}/payment/
    /bookings/{bid}/rooms/{rid}/
    /bookings/{bid}/payment/pconfirmation/
    /bookings/{bid}/payment/pwaiting/
    /rooms/{rid}/
    /rooms/{rid}/booking/
    /rooms/{rid}/booking/cancel/
    /rooms/{rid}/booking/payment/
    /rooms/{rid}/payment/pconfirmation/
    /rooms/{rid}/payment/pwaiting/
```

The REST style requires that all resources should be addressable and connected. Thus, we require that our resource model should not contain an isolated resource. Each resource can be reached from at least one collection resource by navigating one or more associations.

Uniform Interface

A UML class diagram allows us to define a number of operations for each class. Since a RESTful web service provides uniform interface for all resources, all resources would only have from one to four method names GET, POST, PUT, and DELETE. Thus, we do not show operation information in the conceptual resource model. However, by constraining the allowed transition triggers in behavioral model to the standard HTTP method we comply with the uniform interface requirement.

Behavioral Service Model

The purpose of the behavioral model is to describe the behavioral interface specifications of a RESTful web service. It shows the sequence under which operations should be invoked, the conditions under which they can be invoked and the expected results.

We use a UML protocol state machine with state invariants to describe the allowed operations in a web service. A UML protocol state machine is suitable for representing the behavior of a web service as it provides interface specifications that give information about conditions under which methods can be invoked and their expected output.

A UML protocol state machine contains mainly states and transitions. We require that each state has a state invariant that is defined as a boolean expression. We then say that a state is active if and only if its state invariant evaluates to true. A state may contain other states and is called a composite state. In such a case, the actual state invariant of the contained state is given by the conjunction of the state invariant specific for the contained state and the state invariants of all the states that contain it. These state invariants within a composite state should be mutually exclusive. That is, only one state within a region of a composite state can be active at a time.

A transition is an arc from one or more source state(s) to one or more target state(s) labeled with a method name and a guard. If the source states are active, the guard is true and the method is invoked, then the transition may be fired and as a consequence the target state(s) become active. When no guard is shown in the transition it is assumed to be true.

Since we are describing RESTful web interfaces, the only allowed operations are GET, POST, PUT, and DELETE on resources.

The GET method retrieves representation of a resource and it should not have side effects, i.e., not cause a change in the state of the system. Due to the addressability requirement, it is possible to always invoke a GET method over a resource. For example, *GET(/bookings/{bookingId}/payment/)* and *GET(/bookings/{bookingId}/cancel/)* represent GET requests on resources *payment* and *cancel*, respectively. Whenever a GET method is invoked on a resource, it gives the representation of resource as a response if the resource is present, else a response code of 404 is sent back. In practice, the access to resources may be restricted by an authentication and access control mechanism.

The transition triggers can only be defined as POST, PUT, or DELETE operations over resources described in the conceptual model. The POST, PUT, and DELETE methods can have side effects, i.e., they can cause a change in the state of the system.

Our behavioral model shows different states of a RESTful web service and gives information on what HTTP methods on a particular resource can be invoked from a certain state. According to Fig. 5.2, the protocol state machine of HRB service is initiated by the HTTP POST method on the *bookings* resource. The client can make payment for a booking by invoking a PUT method on *payment* resource only if the name of the credit card is same as the name of the guest. The booking service invokes a third party credit card payment service(CCService) from the *paid* state as an internal action. If the CCService is asynchronous, then the booking service invokes a PUT on *pwaiting* resource and the transaction enters a *wait* state. It then invokes a PUT on *pconfirmation* resource when response is received from the CCService. If the CCService is synchronous, the booking service invokes a PUT on pconfirmation resource from the *paid* state when it receives response from the CCService. The case of synchronous and asynchronous services is explained in "Synchronous and Asynchronous Web Services." A booking can be canceled from the composite state *reserve_and_pay* and simple state *pconfirmation_info*. A booking cannot be canceled if it is waiting for the payment confirmation from a third-party service. A booking can be deleted only if it is canceled. Note that all the information needed to process the request on a resource are contained in the invoked method and URL.

A GET method can be invoked on every resource as it is free of any side-effect. However, a closer look at the behavioral model also exposes information about the allowed side-effect methods on a resource. For example, Fig. 5.2 shows that

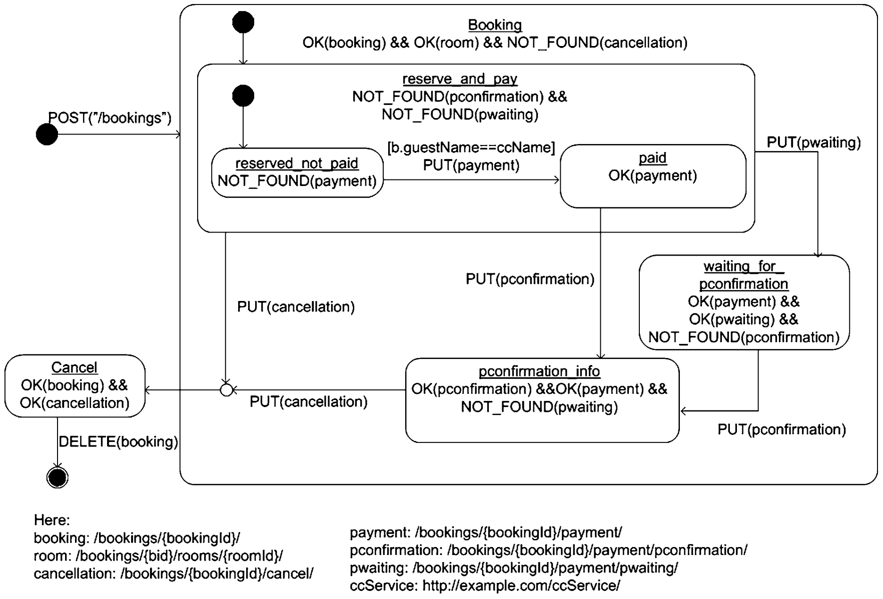

Booking
OK(booking) && OK(room) && NOT_FOUND(cancellation)
reserve_and_pay
NOT_FOUND(pconfirmation) &&
NOT_FOUND(pwaiting)
POST("/bookings")
reserved_not_paid
NOT_FOUND(payment)
[b.guestName==ccName]
PUT(payment)
paid
OK(payment)
PUT(pwaiting)
waiting_for_
pconfirmation
OK(payment) &&
OK(pwaiting) &&
NOT_FOUND(pconfirmation)
PUT(pconfirmation)
PUT(cancellation)
Cancel
OK(booking) &&
OK(cancellation)
PUT(cancellation)
pconfirmation_info
OK(pconfirmation) &&OK(payment) &&
NOT_FOUND(pwaiting)
PUT(pconfirmation)
DELETE(booking)
Here:
booking: /bookings/{bookingId}/
room: /bookings/{bid}/rooms/{roomId}/
cancellation: /bookings/{bookingId}/cancel/
payment: /bookings/{bookingId}/payment/
pconfirmation: /bookings/{bookingId}/payment/pconfirmation/
pwaiting: /bookings/{bookingId}/payment/pwaiting/
ccService: http://example.com/ccService/

Fig. 5.2 Behavioral model for HRB RESTful web service

only a POST (side-effect) method can be invoked on collection resource *bookings*, similarly allowed (side-effect) method on resource *booking*, *payment*, *pwaiting*, *pconfirmation* and *cancel* is PUT. On *booking* resource, a DELETE method can also be invoked.

The guards and postconditions on transitions are defined only using GET requests on request on resources and the request parameters that include values parsed out of the request URI. A guard condition on the transition specifies the condition required to invoke an HTTP method on a resource. For example, consider guard *[b.guestName==ccName]* for the method *PUT(payment)* in Fig. 5.2, where *b* refers to the relative navigation path to resource *booking*. This guard specifies that the PUT method on *payment* resource can be invoked only if the guestName in resource representation of *booking* for booking Id {*bookingId*} matches the name of the credit card provided by the client.

State Invariants Using Resources

State invariants show the current state of an application during the lifecycle of an object. We are representing behavioral interface of a REST web service using protocol state machines. REST invocations do not contain any state or session information, so defining state invariants for REST application states is not obvious.

We address this problem by performing GET requests on different resources and using their representations and response codes to form boolean expressions.

When we invoke an HTTP GET method on a resource, it returns its representation along with the HTTP response code. This response code tells whether the request went well or bad. If the HTTP response code is 200, this means that the request was successful and the referred resource exists. Otherwise, if the response code is 404, this implies that URI could not be mapped to any resource and the referred resource does not exist. We do not treat this 404 code as an error but as an important determinant of protocol state.

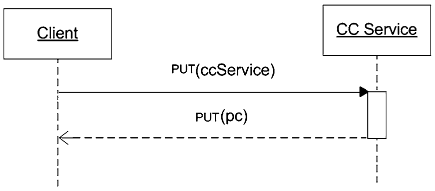
We use a boolean function *OK(r)* to express that the response code of HTTP GET method on a resource *r* is 200. Similarly, the boolean function *NOT_FOUND(r)* is true when the response code of HTTP GET method on resource *r* is 404. These boolean functions on the resources along with the attributes that represent a resource are used to define a state invariant in our RESTful behavioral model.

For example, consider the state invariant for the state *reserved_not_paid* in Fig. 5.2. *NOT_FOUND(payment)* checks the response code for the HTTP GET method on the resource *payment*. It evaluates to true if response code of GET method on *payment* for a particular booking ID({bid}) is 404. For the HRB service to be in state *reserved_not_paid*, the state invariant of this simple state is conjuncted with the state invariants of all the states that contain it, i.e., $NOT\_FOUND(payment)\&\&NOT\_FOUND(pconfirmation)\&\&NOT\_FOUND(pwaiting)\ \&\&OK(booking)\ \&\&OK(room)\&\&NOT\_FOUND(cancellation)$.

Synchronous and Asynchronous Web Services

Interaction between web services can be either synchronous or asynchronous. This interaction is distinguished in the manner request and response are handled. When a client invokes a synchronous services, it suspends further processing until it gets a response from the service. On the other hand, when a client invokes an asynchronous service it does not wait for the response and continues with its processing. The asynchronous service can respond later in time. The client receives this response and continues with its processing.

We have modeled the scenario for both the synchronous and asynchronous third party service in Fig. 5.2. In case of interaction with an asynchronous service, we create a *waiting* state in our state machine. In Fig. 5.2, a third party credit card payment service is invoked when a PUT is invoked on the payment resource. This would invoke CCService as an internal action. If CCService is an asynchronous service, then it may take a long time to process the credit card and confirm the payment back to the client. Thus, the system goes into a wait state for the particular booking with booking ID {bookingId} and resumes processing of other transactions. When a response on payment confirmation is given by the third party service, the processing for this booking is resumed.



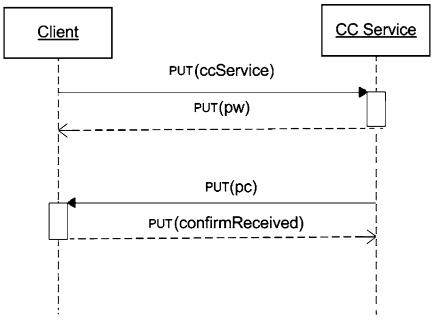


Fig. 5.3 (*Left*) Interaction with Synchronous CC Service. (*Right*) Interaction with Asynchronous CC Service

For synchronous service, there is no need for a *waiting* state since the service does not take long to respond and system can continue with its processing after receiving the response. This is shown in Fig. 5.2 by a direct transition from *paid* state to *pconfirmation_info* state with PUT(pconfirmation) as a trigger.

The two scenarios showing the request and response behavior in synchronous and asynchronous services is shown in Fig. 5.3. The left side shows the scenario in which credit card(CC) verification service is synchronous and on the right hand side show interaction with an asynchronous CC verification service. It may be worth pointing out that the agent PUTing the payment (the client) must also be able to act as a server in order to receive a PUT payment confirmation. As an alternative, the CCService might return 202(Accepted) response with location. This would require the client to poll for confirmation.

Stateless State Machines

We have used state machines to model the stateless behavior of REST web service. Using a state machine to model a stateless interface may seem an oxymoron. In the context of a RESTful service, statelessness is interpreted as the absence of hidden information kept by the service between different service requests. In that sense, a RESTful web service should exhibit a stateless protocol. Also, there is no sense of session or sequence of request in a true RESTful service.

On the other hand, state machines have a notion of active state configuration, that is, what states are active at a certain point of time. If an implementation of an interface described using a state machine would have to keep the active state configuration between different requests, then this would break the statelessness requirement of the RESTful service.

It is notable that the behavioral modeling described above, does not actually require that a service implementation keeps any additional protocol state. In our approach a state is active if its invariant evaluates to true, but the invariants are defined using addressable application resources. Therefore, an implementation of a service can determine the active state configuration by querying the application state. There is no need to keep any additional protocol state.

Determining what is the active state configuration of the interface state machine every time that a service implementation has to fulfill a request may be a slow task in the case of complex interfaces with many states. However, in practice it is not necessary to explore all states in the state machine but only the source states of the transitions that can be triggered based on the current request. We show in the next section how we can do that by computing the precondition (and postcondition) of each method request.

Service Preconditions and Postconditions

In this section, we show how to extract the contract information from a UML protocol state machine with state invariants. The contract contains the precondition and postcondition for each method that triggers a transition in the behavioral model.

The precondition of a method states under what conditions a method can be triggered. We say that the precondition of a method *m* is satisfied when the state invariants of all the source states of transition *t* are true along with its guard condition.

In a similar manner, if a method *m* triggers a transition *t* in a behavioral model, then its post-condition is satisfied when the state invariants of all the target states of transition *t* are true along with the postcondition annotated on the transition *t*.

In order to shorten the description of the contract we use path variables to represent the address of a resource. First, the precondition for a method that triggers a transition in the behavioral model is presented. The precondition of a method *m* is given by taking into account all the transitions that are triggered by *m*. If it is a simple transition, then the state invariant of its source state is conjuncted with the guard of the transition. In case the transition is a trigger to more than one transition, with true guards, and all the transitions have different source states, then the precondition is given by taking a disjunction of state invariants of all the different source states. This implies that the method can trigger a transition whenever it is in one of its source states.

A transition can occur from one state to another if the method that triggers this transition is invoked and its precondition is true. For the transition to be successful, the postcondition of the transition should also be true after the method is invoked. This is specified by the *implication* operator that relates a precondition of a transition with its postcondition.

A postcondition for a method is extracted from the protocol state machine by manipulating the state invariants of the target states of transitions and the post-conditions on transitions. The post-condition of a fork transition, with true postcondition, specifies that the state invariants of all its target states are true and for a self-transition, its post-condition ensures that the same state invariants are true that were true before invoking the HTTP method.

For the details and formal definitions of generating preconditions and postconditions for different elements in a UML protocol state machine of a class readers are referred to Porres and Rauf (2010).

The postcondition of a transition will be evaluated only if the precondition for that transition is true. We define as *pre_OK(r)* the function that gives boolean value of *OK(r)* on resource *r* before invoking the trigger method. Similarly, *pre_b.guestName* and *pre_NOT_FOUND(r)* give the representation of *booking* and boolean value of *NOT_FOUND(r)* before invoking the trigger method, respectively.

The excerpt below from the list of high-level contracts generated from Fig. 5.2 shows the contracts generated for the HTTP method PUT on *payment* resource.

```
PATH
 b: bookings/{bid}/
 r: bookings/{bid}/rooms/
 p: bookings/{bid}/payment/
 pc: bookings/{bid}/payment/pconfirmation/
 pw: bookings/{bid}/payment/pwaiting/
 c: bookings/{bid}/cancel/

PUT {bookings/{bid}/payment/}
 precondition
  ((OK(b) && OK(r) && NOT_FOUND(c)) &&
  (NOT_FOUND(pc) && NOT_FOUND(pw)) && NOT_FOUND(p) &&
     [b.guestName == ccName])

 postcondition
  ((pre_OK(b) && pre_OK(r) && pre_NOT_FOUND(c)) &&
  (pre_NOT_FOUND(pc) && pre_NOT_FOUND(pw)) && pre_NOT_FOUND
      (p) && [pre_b.guestName == ccName]) ==>  ((OK(b) && OK
      (r) && NOT_FOUND(c)) &&
  (NOT_FOUND(pc) && NOT_FOUND (pw))&& OK(p))
```

The conceptual model as shown in Fig. 5.1 and behavioral model as show in Fig. 5.2 are implemented as REST web services. This is explained further in the next section.

Implementation of a Service Using the Django Framework

Django is a web framework that makes it easy to develop web applications and web services in Python. At a glance, Django can be understood with its three basic files that support separation of concerns, i.e. models.py, urls.py and views.py where models.py contains descriptions of database tables, views.py contains the business logic, and urls.py specifies which URIs map to which view. For a

```
from django.db import models

class room(models.Model):
    rType = models.CharField(max_length=200)
    floor = models.IntegerField()

class guest(models.Model):
    fName = models.CharField(max_length=200)
    phone = models.IntegerField()
    email =  models.CharField(max_length=200)

class booking(models.Model):
    bDate =  models.DateTimeField()
    cancel = models.BooleanField(default=False)
    cancel_note = models.CharField(max_length=500)
    room = models.ForeignKey(room)
    gName = models.CharField(max_length=500)

class payment(models.Model):
    amount = models.FloatField()
    pDate = models.DateTimeField()
    confirm = models.BooleanField(default=False)
    waiting = models.BooleanField(default=False)
    p_try = models.IntegerField(default = 0)
    ccName = models.CharField(max_length=500)
    booking = models.ForeignKey(booking)
```

Listing 5.1 Implementation of Database Models for HRB Service

detailed working of Django Framework, readers are encouraged to read Django Documentation (Django Software Foundation 2010) and Django Book (Holovaty and Kaplan-Moss 2010).

The design approach we have used to design REST web services in this chapter can be easily implemented in Django. In this section, we show how this implementation is done. We carry forward the example of HRB service demonstrated above and show its implementation procedure.

The main steps in our implementation phase are:

- Implement database tables in models.py
- Create views for each resource and its transitions in views.py
- Map relative URIs from resource model to respective views in urls.py.

As a first step, the database tables are specified in models.py. The database tables we have created are shown in Listing 5.1.

In the second step, for each resource, shown in the conceptual resource model, a view is defined. The information on *allowed* and *not-allowed* methods is retrieved from behavioral model. The incoming request to the view is verified against the allowed methods and redirected to the view that supports the request method for the resource.

```
def booking_payment(request, bid):
    if not request.method in ["GET", "PUT"]:
        return HttpResponseNotAllowed(["GET", "PUT"])
    if request.method == "GET":
        bid = bid
        return booking_payment_get(request, bid)
    if request.method == "PUT":
        bid = bid
        amnt = request.POST.get('amnt')
        ccName = request.POST.get('ccName')
        return booking_payment_put(request, bid, amnt, ccName
            )

def booking_payment_get(request, bid):
        p = payment.objects.filter(booking=bid)
        if p:
            json = serializers.serialize("json", p)
            return HttpResponse(json, mimetype="application/
                json")
        else:
            return None

def booking_payment_put(request, bid, amnt, ccName):
        b = booking_detail_get_local(bid)
        r = room_detail_get_local(bid)
        c = booking_cancel_get_local(bid)
        p = booking_payment_get_local(bid)
        pc = booking_pconfirmation_get_local(bid)
        if not p:
           pre_p = False
        else:
           pre_p = True
        deserialized = serializers.deserialize("json", b)
        b_detail = list(deserialized)[0].object
        a = []
        for field in ["bDate", "cancel", "cancel_note", "room"
            , "gName"]:
            new_val = getattr(b_detail, field, None )
            a.append(new_val)
        if  b and r and not p and not pc and not c and a[4]==
            ccName:
            now = datetime.datetime.now()
            cc = ccName
            a = amnt
```

Listing 5.2 Payment View

The first view *booking_payment*(*request*, *bid*) in Listing 5.2 shows implementation of *payment* resource. The behavioral model in Fig. 5.2 shows that the allowed methods for this resource are GET and PUT. These two methods are listed in the

```
            p = payment(confirm=False, pDate=now, waiting=
                False, amount=a, p_try=0, ccName = cc,
                booking_id=bid)
            p.save()
        b = booking_detail_get_local(bid)
        r = room_detail_get_local(bid)
        c = booking_cancel_get_local(bid)
        pc = booking_pconfirmation_get_local(bid)
        post_p = booking_payment_get_local(bid)
        if  b and r and not pre_p and post_p and not pc and
            not c:
            response = HttpResponse("created")
            response.status_code = 201
            return response
        else:
            response = HttpResponse("not created")
            response.status_code = 406
            return response
```

Listing 5.2 (continued)

```
urlpatterns = patterns('',
                        (r'^bookings/$', collection_bookings),
                        (r'^bookings/(\d{1,3})/$',
                            booking_detail),
                        (r'^rooms/$', collection_rooms) ,
                        (r'^bookings/(\d{1,3})/rooms/$',
                            room_detail),
                        (r'^bookings/(\d{1,3})/payment/$',
                            booking_payment),
                        (r'^bookings/(\d{1,3})/payment/waiting
                            /$', booking_waiting),
                        (r'^bookings/(\d{1,3})/payment/
                            pconfirmation/$',
                            booking_pconfirmation),
                        (r'^bookings/(\d{1,3})/cancel/$',
                            booking_cancel),
)
```

Listing 5.3 Relative URIs and views mapping for HRB Service

list of allowed methods in *booking_payment* view and each incoming request to this view is first verified to be one of these methods, otherwise an HTTP response of method not allowed is given.

In the third step, the relative URIs shown in the conceptual resource model are mapped to the respective views. Every resource in our conceptual model is addressable. We can get the relative URI for each resource directly from Fig. 5.1 that is then mapped to the respective views as show in Listing 5.3.

Users can use cURL to invoke URIs specifying the methods they want to invoke on the service. cURL is a command line tool that is a capable HTTP client and supports most of HTTP methods, authentication mechanisms, headers etc. (cURL 2010). For invoking a POST method on *payment* resource with *amnt* value, on local server, the following command can be used on cURL:

$curl - X\ PUT - d\ amnt = 115\ -d\ ccName =''\ Thomas''\ http : //127.0.0.1 : 8000/\ bookings/3/payment/$

Now lets look in detail on the implementation of views. A separate view is implemented for each of the allowed methods on each resource. Once a view related to a specific URL is called, it further redirects the control to the view that corresponds to the invoked HTTP method.

As an example, we are only looking into the *payment* resource and its allowed methods in Listing 5.2. The allowed methods on *payment* resource are GET and PUT as specified in Fig. 5.2. When the client invokes */bookings/13/payment/*, control is passed to *booking_payment* view. This view verifies the input method and if the request method is neither GET nor PUT, an HTTP not allowed response is given. If the method is GET or PUT on *payment*, the client is redirected to *booking_payment_get*(*request*, *bid*) view or *booking_payment_put*(*request*, *bid*, *amnt*, *ccName*) view, respectively.

The GET view, i.e., *booking_payment_get*(*request*, *bid*), queries the database, retrieves the payment information for booking id 13 and returns it as a JSON object. If there is no booking record with id 13, then a response code of 404 is returned.

The PUT view creates the specific resource and returns a successful HTTP response method. When the client invokes */bookings/13/payment/* with PUT method, the control goes to *booking_payment_put*(*request*, *bid*, *amnt*, *ccName*) view and a payment record is entered for booking with id 13.

However, if a payment record is already present for this booking id, then the operation of inserting additional record in payment table should not be executed. Such rich behavioral specifications are present in the behavioral model and earlier in "Service Preconditions and Postconditions" we saw how preconditions and postconditions of methods can be generated from this model. We now detail how these behavioral specifications are inserted for methods in Django Web Framework.

The pre-condition of a method is extracted from the state machine by manipulating the state invariants of all the source states and guard on the transition. Likewise, a post-condition is extracted by manipulating the state invariants of all the target states and post condition on the transition.

When a method with side-effects, i.e. PUT, POST or DELETE is called on a resource, we need to extract the current state of different resources to check whether the conditions to invoke the method are satisfied. In a similar fashion, we have to check the status of different resources to ensure that desired effect is created before returning the client a success message. By current state we mean the presence or absence of a resource or values of its attributes at the time of invoking certain method.

In Django, we extract the current state of resources by calling the view that maps to GET request on the resource. However, to take advantage of relative URI mechanism and to reduce the number of HTTP calls, local GET views are

```
def booking_payment_get_local(bid):
        p = payment.objects.filter(booking=bid)
        if p:
            data = serializers.serialize("json", p)
            return data
        else:
            return None
```

Listing 5.4 Excerpt of Local GET View on 'payment' for HRB Service

implemented for each resource. The local GET views retrieve information from the database and return them as normal objects rather than as HTTP response objects. An implementation of local GET view on *payment* resource is shown in Listing 5.4.

The pre and post conditions are asserted in each of the views that correspond to the methods that trigger a transition in state machine. Listing 5.2 shows how pre and post conditions are asserted for PUT method on *payment*. Information of the resources that form the state invariant of source states and guard condition is stored in different variables. These variables are combined as a boolean expression and asserted as an *if* condition before performing the desired task. Similarly, before giving a success response to the client, a local get is performed on the resources that make the state invariant of target states and transition's post conditions. Only if the expected behavior is observed, a success response is given to the client.

Implementation of a Service Monitor

A service monitor can be used to continuously verify the functionality of an implemented web service. This monitoring mechanism can keep a check on the behavior of both the client and the provider. The client is checked for invocation to the service under right conditions and the provider of the service is constraint to provide the implementation as specified.

The monitoring mechanism can be implemented in Django by using the rich behavioral information present in our state machine. The service monitor is implemented as a service proxy. It listens for requests from the client, verifies the conditions to invoke the method and then forward it to the actual service implementation.

The behavioral model provides a rich behavioral interface that can be published with the service as a specification. This gives information about the conditions in which a method should be invoked on its interface and also about its expected conditions. This specification of a service interface can be used to build a proxy interface to test the functionality of that service and to invoke the service in right conditions.

```
def booking_payment_get(request, bid):
    print "booking payment get"
    req = urllib2.Request('http://127.0.0.1:8000/bookings/%s/
        payment/' % bid)
    try:
        response = urllib2.urlopen(req)
        the_page = response.read()
        return HttpResponse(the_page)
    except:
        return HttpResponse(status=404)
```

Listing 5.5 Excerpt of GET view in Proxy Interface

```
def booking_payment_put(request, bid, amnt, ccName):
   b = booking_detail_get(request, bid)
   r = room_detail_get(request, bid)
   c = booking_cancel_get(request,bid)
   p = booking_payment_get(request, bid)
   pc = booking_pconfirmation_get(request, bid)
   pw = booking_pconfirmation_get(request, bid)
   if not p.status_code == 200:
       pre_p = False
   else:
       pre_p = True
   if  b.status_code = 200 and r.status_code == 200 and p.
       status_code == 404 and pc.status_code == 404 and pw.
       status_code == 404 and c.status_code == 404:
       values ={'amnt': 33, 'ccName': 'Thomas'}
       mydata = urllib.urlencode(values)
       opener = urllib2.build_opener(urllib2.HTTPHandler)
       request = urllib2.Request('http://127.0.0.1:8000/
           bookings/%s/payment/' % bid, data=mydata)
       request.add_header('Content-Type', 'your/contenttype')
       request.get_method = lambda: 'PUT'
       url = opener.open(request)
   else:
       return = HttpResponse(status=404)
   post_p = booking_payment_get(request, bid)
   if  b.status_code = 200 and r.status_code == 200 and pc.
       status_code == 404 and pw.status_code == 404 and c.
       status_code == 404 and not pre_p and post_p.status_code
        == 200:
       return HttpResponse(the_page,status=201)
   else:
       return HttpResponse("not created",status=406)
```

Listing 5.6 PUT Method on Payment in the Proxy Interface

In this section, we show how we have implemented a proxy interface for HRB service detailed above. In proxy interface, a method is implemented for each of the methods that are invoked on the REST web service interface using urllib2. urllib2 is a Python module that is used to fetch URLs (urllib2 extensible library for opening URLs 2010). In a proxy interface for HRB service, a GET method on payment resource is implemented as shown in Listing 5.5.

Each GET view returns an HTTP response object. When a POST, PUT or DELETE method is implemented in the proxy interface, it manipulates the status codes of the HTTP response objects and asserts them as method pre and post conditions. An excerpt of HRB proxy interface that shows a PUT method on the payment resource is given as shown in Listing 5.6.

Conclusions

RESTful web services can be used in rich services that go beyond simple operations of creating, retrieving, updating, and deleting data from the database. These rich services should also offer interface that would exhibit REST features of uniform interface, addressability, connectedness, and statelessness. In this chapter, we discuss the design methodology that creates RESTful web services by construction. The approach uses UML class diagram and state machine diagram to represent the structural and behavioral features of a REST web service. The conceptual resource model that represents the structural feature adds addressability and connectivity features to the designed interface. The uniform interface feature is offered by constraining the invocation methods in the state machine to HTTP methods. In addition, to provide the feature of statelessness in our interface we use a state machine for behavioral modeling. This oxymoron is addressed by taking advantage of the fact that state invariants can be defined using query method on resources and the information contained in their response codes.

The rich behavioral specifications present in the behavioral model show the order of method invocations and the conditions under which these methods can be invoked along with the expected conditions. We use this behavioral model to generate contracts in the form of preconditions and postconditions for methods of an interface.

The design approach is implemented in Django web framework and the contracts generated from the behavioral model are asserted as contracts in the implemented interface.

A proxy interface is also implemented in Django as a service monitor. This service monitor can also be implemented for services that are already implemented and only provide their behavioral specifications in natural language or in any other form. A service monitor can continuously verify functionality of a service and reports if a service user violates a precondition or the implementation does not provide the expected behavior.

References

cURL. 2010. http://curl.haxx.se/.

urllib2 extensible library for opening URLs. *Python Documentation*, 2010. http://docs.python.org/library/urllib2.html.

Django Software Foundation. Django Documentation. *Online Documentation of Django 1.2*, 2010. http://docs.djangoproject.com/en/1.2/.

A. Holovaty and J. Kaplan-Moss. The Django Book. *Online version of The Django Book*, 2010. http://docs.djangoproject.com/en/1.2/.

T. Mens and P. V. Gorp. A Taxonomy of Model Transformation. *Proceedings of the International Workshop on Graph and Model Transformation*, 2005.

I. Porres and I. Rauf. From uml Protocol Statemachins to Class Contracts. *Procceedings of the International Conference on Software Test, Verification and Validation(ICST 2010)*, 2010.

I. Porres and I. Rauf. Modeling Behavioral RESTful Web Service Interfaces in UML. *Accepted for Publication in 26th Annual ACM Symposium on Applied Computing Track on Service Oriented Architectures and Programming (SAC 2011)*, 2011.

OMG UML. 2.2 Superstructure Specification. *OMG ed*, 2009. http://www.omg.org/spec/UML/2.2/.

Chapter 6
Quantifying Integration Architectures

Jan Algermissen

Abstract The products or services offered by enterprises today increasingly depend on information products realized by the corporate IT department. Often the time to market of a product is significantly affected by the time it takes to realize its IT-enabled aspects. In this regard, minimizing realization time within the IT department often becomes the essential factor for bringing a given product to market earlier than the competition.

This chapter proposes a methodology for determining a measure of how the integration styles of given IT systems affect the ability of these systems to adapt to changing requirements.

Introduction

The products or services offered by enterprises today increasingly depend on information products realized by the corporate IT department. Often the time to market of a product is significantly affected by the time it takes to realize its IT-enabled aspects. In this regard, minimizing realization time within the IT department often becomes the essential factor for bringing a given product to market earlier than the competition.

Realization of new functionality to be delivered by an existing IT system usually involves changing system components that are already deployed and interacting with each other to support current business processes. The amount of time and resources required to change these existing components is to a large extend determined by the amount of coupling between them and whether the kind and amount of coupling matches the given integration situation. For example, tight

J. Algermissen (✉)
NORD Software Consulting, Kriemhildstrasse 7, 22559 Hamburg, Germany
e-mail: algermissen@acm.org

coupling between a provider and one single consumer is not a critical situation whereas tight coupling between a provider and a million consumers on the World Wide Web is a changeability disaster.

Given this situation, it appears useful to provide a measure for the overall integration architecture quality of an IT system that expresses how well its component connections match the circumstances of their use. Such a measure would provide a way to assess an IT department's overall resistance to evolution. It could be used to compare integration architectures or could be tracked over time to verify the success of an integration architecture management strategy.

This chapter proposes a methodology to define such a quality measure in terms of the coupling created by the applied connectors on the one hand and in terms of the specific circumstances of their use on the other.

Kinds of Change Impact

When components consume capabilities provided by other components dependency relationships are created. Changes applied to providers potentially affect the consumers and therefore have to be considered during implementation design and deployment planning.

As the basis for a detailed change impact analysis the following sections provide a classification of the various ways in which changes to one component can affect other, depending components.

The kinds of change impact are ordered according to increasing perceived complexity and cost.

Terminate or Suspend Application

Changing the functionality provided by a given component requires the deployment of a new software release realizing the associated changes. Deployments result in a temporary unavailability of the changed components. Consumers are affected by provider unavailability if the applied connectors do not provide temporal decoupling between components.

If temporal coupling exists the unavailability of the provider must be coordinated with the owners of the consumers. Depending on their nature it will be required to suspend or terminate the applications that use the affected components. For example, in automated supply chain management it could be necessary to suspend the submission of new orders and to wait for active ordering processes to terminate before deploying a new version of a provider-side component.

Configure, Build, and Deploy Consumer

To augment the capabilities of a provider it can be necessary to change aspects that do not directly affect concerns of the consumer implementation but require a configuration update or upgrading a software library. If the connector used does not protect a consumer from changes of this kind it will be necessary to reconfigure or rebuild and subsequently to re-deploy the consumer component.

Re-deploying the consumer component not only requires termination of the affected applications; consumer-side software release processes and deployment schedules lead to additional complexity for coordinating the change.

Commonly experienced consumer-side configuration- or build impacts are, for example, the need to update a configured provider address or to upgrade a database driver library.

Data Format Change

New requirements often lead to an evolution of the formats used to transfer data between communicating components. If the applied connectors neither make it possible to negotiate between new and previous data-format versions nor enable the transparent use of transforming intermediaries consumers must also be changed to be able to process the new data format.

In this case, data format changes require implementation activities on the consumer side which lead to additional coordination cost between consumer and provider owners beyond the cost of application termination and component re-deployment.

The complexity of data format changes ranges from simple (for example, adding a field to an address data structure) to highly complicated (for example, changing the data model of a data warehouse, impacting the SQL statements of hundreds business reports).

Connector Protocol Change

Connectors that are realized as a set of procedures often have an associated set of assumptions about the order in which the procedures are called or require relations between parameter values. Such rules are effectively protocols for the interaction and the communicating components must share an understanding of these protocols (Shaw and Garlan 1996).

When the provider changes the rules for the interaction (for example, adds a method that must be called before a given other method) the consumers must also be changed to account for the altered protocol.

Like changes to the data format Connector Protocol Changes result in a requirement to change the consumer implementation unless the connector used prevents protocol changes from affecting the consumer.

(REST's hypermedia constraint (Fielding 2000, pp. 100) is an example of how protocol coupling can be removed from the connector interface).

Shared Identity Change

Components use identifiers to refer to other components and the data and control elements they expose. Some architectural styles remove the need for identity from the connector interface [for example, Event-Based Integration (Fielding 2000 p. 55)] others provide dynamic identifier resolution (for example, DNS) or runtime identifier discovery (for example, hypermedia) to reduce the coupling created by shared identify.

In the absence of such architectural strategies, changes to one component can lead to a requirement to update the shared identity maintained in other components. This can result in component configuration changes or programming activity if identity issues are interweaved with source code.

Common examples of Shared Identity Changes are adjusting database connection settings, changing relational table- or column names, or updating resource identifiers of HTTP-based services that expose a fixed set of URIs as part of their service description.

Communication Model Change

Some connectors do not specify how coordination is achieved between the processes of the communicating components or how communication failures are detected and handled. Instead, these aspects of the communication are deferred to the applications using the connector.

Changes to one component can significantly change the communication model and therefore lead to extensive design and implementation changes of other components.

In the case of components that communicate using a shared file, for example, concurrency control might be changed from a file locking based approach to explicit timing assumptions (component A writes at 9 AM and component B reads at 4 PM).

The possibility of communication model changes is typically found in ad-hoc enterprise integration approaches such as communicating through a shared file or database table. It is rarely (if at all) found in "modern", research-based connectors.

Table 6.1 Kinds of change impact

Change impact kind	Description
Terminate or suspend application	Is it possible that a change to one component requires termination or suspension of applications?
Configure, build, and deploy consumer	Is it possible that a change to one component requires other components to be re-deployed?
Data format change	Is it possible that a change to one component requires other components to update their implementation regarding data structure assumptions?
Connector protocol change	Is it possible that a change to one component requires other components to update their implementation to adjust to a protocol change, for example regarding the sequence of operations?
Shared identity change	Is it possible that a change to one component requires another component to change identifiers of data items or the changed component itself?
Communication model change	Is it possible that a change to one component requires other components to change the model in which they interact with the component? For example, to change from synchronous to asynchronous interactions?
Programming language change	Is it possible that a change to one component requires other components to be re-implemented in another programming language?

Programming Language Change

Certain technical realizations of several connectors prescribe the programming language (or the programming language environment) for component implementations. If, for technical or other reasons, the programming language environment of one component changes the other components must be re-implemented in a language of the new environment. The impact of such a change can be extremely broad, potentially making the change itself impossible.

The possibility of a programming language change is not a property of a given connector but rather of individual connector implementation technologies. The methodology proposed in this chapter differentiates between general connector properties and technology specific effects.

Summary of Kinds of Change Impact

Table 6.1 summarizes the kinds of change impact discussed above.

Connectors

The connector (Fielding 2000, pp. 10) used to mediate communication between components determines the amount of coupling between them. I will illustrate the development of the methodology using a set of connectors typically found in enterprise integration scenarios.

Some of the connector names below also are the names of the corresponding architectural styles. However, this chapter focuses on the connectors and the amount of change potentially exposed on connected components.

File Transfer

A File Transfer connector (Hophe and Woolf 2004a) mediates communication between components through one or several files residing in a well-known location. The components exchange data by reading and writing to these files. Process coordination is usually achieved by file naming conventions (for example, adding a suffix like*.done* to already processed files) or the presence or absence of flag files.

Except for the basic file I/O operations provided by the underlying operating system all aspects of the communication must be specified by the application using the connector. This increases coupling because all participating components have to implement the specific coordination semantics.

The notion of File Transfer Connector also applies to scenarios where some or all of the components access the shared file(s) using FTP (a typical solution for integrating with news feed providers).

Shared Database

A Shared Database connector (Hophe and Woolf 2004b) mediates communication through tables managed by a relational database system. The components exchange information by reading and writing to known tables. Similar to the *File Transfer* connector all communication aspects beyond the interaction with the database are deferred to the application using the connector. This results in tight coupling, especially because the component implementations share knowledge about the table and column names, the specific coordination model and potentially specifics of the database product used.

The use of Shared Database Connectors is very common in the financial industry, especially on the basis of data warehouse tables (sometimes leading to operational use of data gathered for analytical purposes) or on the basis of the relational persistence layer of products (instead of using the programming interface provided by the product).

Remote Procedure Call

Remote Procedure Call (Birrell and Nelson 1984) connectors mediate communication by invoking named procedures over a communication network. Remote Procedure Call connectors employ a request/response communication model. Calling a remote procedure transfers control and data from one component to another for the duration of the call. Distributed objects styles also use Remote Procedure Call connectors.

Common implementation technologies are XML-RPC (Winer 1999), Java RMI (Oracle 2009), CORBA (OMG 2008), and DCOM (Microsoft 2007). The early approach to Web services also favored an RPC-style approach (Tilkov 2005).

The primary coupling concern raised regarding Remote Procedure Call connectors is that all components must share an understanding of the specific connector protocol. Other concerns are decreased understandability because application state is distributed among the communicating components and potential performance problems.

Of particular interest in the context of unRESTful uses of REST-based architectures is the realization of Remote Procedure Call connectors by tunneling the remote method invocation through HTTP requests (Algermissen 2010a).

HTTP Type I

HTTP Type I connectors (Algermissen 2010b) constitute the most common form of architecturally incorrect use of HTTP. The server component typically specifies a set of URIs that the client has to know to communicate with the server. In addition to the set of URIs, the returned and accepted message formats are specified as well as the service specific status codes to expect. Typically, this information is provided in a human-(HTML) or machine-readable form [WADL (Hadley 2009)].

While the HTTP methods are used according to their specification there exists tight coupling between provider and consumer around the service aspects described in the service description document. The amount of coupling is comparable to the amount of coupling in the case of Remote Procedure Call connectors.

A prominent example of a service exposing this kind of connector is the Twitter API (Twitter 2009).

Message Oriented Middleware

Message Oriented Middleware connectors enable components to communicate through the exchange of asynchronous messages. Usually Message Oriented Middleware implementation technologies provide the property of reliable message delivery and thus temporal decoupling between components.

Message Oriented Middleware constrains the component interface to a single method with the semantic of "processThis" (Baker 2005).

Examples of Message Oriented Middleware connectors are Java Message Service (Sun 2008), Microsoft MQ (Microsoft 2009), Document-Oriented Web Services (Baker 2005), and the Simple Mail Transfer Protocol (Klensin 2001).

Event-Based Integration

In the Event-based Integration style (Fielding 2000, p. 55), components communicate through event publishing. Providers announce or broadcast (usually typed) events and the event system will invoke consumer components that have registered for certain types of events.

A key property of event-based integration is decreased coupling because the communicating components do not need to be aware of each other's identity. In addition, event-based integration provides temporal decoupling if the underlying messaging technology provides reliable messaging.

Common implementation technologies for Event-Based Integration connectors include Java Message Service (Sun 2008), WS-Eventing (Box et al. 2006), XMPP Publish-Subscribe (Millard et al. 1999), PubSubHubbub (Fitzpatrick and Slatkin 2010).

HTTP Type II

HTTP Type II (Algermissen 2010c) connectors are another common misuse of HTTP. HTTP Type II is an improvement compared to HTTP Type I because it exposes less service specific information and thereby reduces coupling. HTTP Type II uses only HTTP standard status codes and specific media types.

Nevertheless, similar to HTTP Type I, the URI space, the accepted and returned media types as well as the predefined set of possible response status codes are specified by a design time artifact (for example, a WADL document).

The possible change impacts are reduced because HTTP Type II connectors do not expose specific data formats and can take advantage of HTTP redirection and to some extend content negotiation.

An example of a service that exposes this kind of connector is the Google Calendar API (Google 2010).

REST

The REST architectural style (Fielding 2000) emphasizes overall system simplicity and decoupling of components. Components communicate through the exchange of self-describing messages in a request–response interaction style. Uniform connector semantics, self-describing messages and the runtime-only exposure of control structures (Fielding 2000, pp. 100) remove all coupling between individual clients and servers.

HTTP 1.1 (Fielding et al. 1999) is the implementing technology of REST.

Change Impact Potential of the Connectors

In the case of most connectors, changes to provider components have an impact on their consumers. However, neither imposes every connector the same kinds of impact nor requires every kind of impact the same amount of resources to address it. To achieve a realistic comparison between connectors, it is essential to examine which connectors potentially lead to which kinds of impact and whether a given impact is very likely to occur or is just an architectural possibility.

The following table shows for every examined connector the possible change impacts. The symbol *o* indicates that it is architecturally possible that the given change impact occurs; that it is not possible to entirely protect consumers from this impact and still apply any kind of desired change to the provider. The symbol + indicates that the given change is relatively common and the symbol ++ indicates that the given change impact is very likely to occur, for example, because it is the usual consequence of the best practices when changing the provider.

The absence of a symbol indicates that a connector is inherently capable of enabling any kind of new provider capability without imposing the given impact on any consumer (Table 6.2).

Table 6.2 Change impact potential of examined connectors

	Terminate or suspend application	Configure, compile, and deploy consumer	Data format change	Connector protocol change	Shared identity change	Communication model change
File transfer	++	++	++	++	++	++
Shared database	++	++	++	++	++	++
Remote procedure call	++	++	+	++	+	
HTTP Type I	++	++	++	+	++	
Message oriented middleware	+	+	+		o	
Event-based integration	+	+	+		o	
HTTP Type II REST	o	o		o	o	

The most problematic connectors are *File Transfer* and *Shared Database*. Both cannot protect the communicating components from any kind of change impact. In addition, all change impacts are equally likely to occur since there are no patterns or best practices that make any of the changes less likely. These connectors are closely related to an "ad-hoc" or "do what works" problem solving style – when new requirements arise, the necessary changes are made without architectural guidance.

Remote Procedure Call and *HTTP Type I* show similar impact potential. *Remote Procedure Call* emphasizes communication through method semantics and change impact tends more towards connector protocol changes. *HTTP Type I* on the other hand emphasizes communication through data format semantics and impact potential tends more towards data format changes.

Message Oriented Middleware and *Event Based Integration* generally have less impact potential due to their ability of temporal decoupling and their focus on uniform method semantics. Both make it less likely that a possible change actually impacts a consumer.

HTTP Type I, *Message Oriented Middleware* and *Event Based Integratio* all focus on data centric communication. However, use of standard document formats, for example, UBL (Bosak and McGrath 2006) seems to be more widely practiced with the latter two than with *HTTP Type I* based designs.

The impact potential of *HTTP Type II* is even lower because the possibility of HTTP redirect- and content negotiation mechanisms makes it much easier to realize new provider functionality without impacting consumers in any way. However, impact is still possible because coupling between provider and consumer exists around the exposed service description.

REST is the only connector that makes provider evolution completely independent from consumer impact. Any kind of evolution can be achieved without causing any impact on consumers. This is achieved by architecturally removing the need for any service specific descriptions. Without such descriptions all coupling between consumer and provider is removed, enabling impact-free evolution.

The Significance of Component Usage

So far I have presented an approach to capture and compare the effect that connectors have on the amount of coupling between components. However, the coupling created by a connector is not sufficient to determine how difficult it is to change the providing component. We also need to consider how many consumers are connected through a given connector and how easy it is to coordinate change activity with them. It is the combination of both, connector coupling and the circumstances of connector use that determine the effect on the consumer.

If the applied connector matches the circumstances of its use the effect on the changeability of the provider will be within reasonable bounds but a connector mismatch can effectively make provider evolution impossible.

For example, tight coupling between a provider and a single consumer that are owned by a single authority is usually not considered an obstacle to evolving the provider component. On the other hand, any kind of coupling between a provider and many consumers that are owned by authorities beyond the provider's control present a situation where changing the provider must be considered impossible.

In the following sections, I will introduce the concepts of *Agency Distance* and *Consumer Cardinality* to denote how easily coordination between component owners can be established, and to express how many consumers are tied to a provider using a given connector.

Agency Distance

Components are owned and operated by human authorities. An agency boundary (Khare and Taylor 2004) denotes the set of components that are owned by a single authority and for which consensus and coordination can be enforced. When a change to a provider component imposes a corresponding change to its consumers the component owners must coordinate their related adaptation activities. The overall cost (and likelihood) of achieving this coordination depends on which kind of connector has been used and also on whether the component owners reside within the same agency boundary or not.

Because of this significance of how far apart the component owners are I use the term *Agency Distance* to capture whether one must cross an agency boundary when connecting components.

The owners of any two communicating components can reside within a single agency boundary or within separate ones. For the latter case, two possibilities exist: the separate agencies can either be rather closely related, for example, as business partners, or almost completely unrelated as are, for example, an online retailer and its potentially millions of customers.

Therefore I differentiate the three agency distances shown in Table 6.3.

Table 6.3 Agency distances

Name	Description	Example
Same	Communicating components are owned by the same authority.	An application connected to its private database, integrated business systems of a single organizational unit.
Near	Communicating components are owned by different agencies, but coordination between the agencies is possible though often undesirable. Especially when integrating with business partners.	Data warehouse integration between subsidiaries and a higher entity, Supply chain management between business partners, e.g. suppliers and manufacturers.
Far	Communicating components are owned by different agencies. Coordination between the agencies is either impossible or highly undesirable.	Online retailer and its customers, a payment gateway offering its service to its customers, a content provider offering its services to subscribers worldwide.

Table 6.4 Consumer cardinalities

Name	Description	Example
0	No consumers at all.	A provider component not yet taken into production or not yet made accessible.
1	Exactly one consumer. No expectation or intention that there will be other consumers.	Two systems connected for synchronization. An application and its "private" database.
n	Very few (e.g. <4), exactly known consumers, no expectation or intention that this number will significantly increase	Inventory system used by a few other business applications.
N	Many, usually unknown, uncontrollable consumers, coordination impossible or very expensive.	Online retailer and its customers, a data warehouse and business systems that use it, a system that supports on-the-road salesmen.

Consumer Cardinality

A dependency relationship exists between provider- and consumer components. The more consumers use a given connector the more difficult it becomes to change the provider because the impact of the change needs to be coordinated with the owners of the consumers. An extremely large number of consumers belonging to many different owners makes it effectively impossible to coordinate a change.

We can differentiate three kinds of consumer cardinalities with regard to how strongly they affect a provider's ability to change (Table 6.4).

Especially in enterprise integration contexts the consumer cardinality n has a strong tendency towards changing to a cardinality of N because it is often the case that it is not exactly known how many system actually depend on a given provider. There are, for example, situations when changing a data warehouse is practically impossible because several hundreds of business reports directly depend on the database schema. Thus, it is important to emphasize that the consumer cardinality of N does not only apply to large scale, public services on the World Wide Web but is frequently found inside enterprise IT systems.

Connector Usage

The two preceding sections introduced the concepts of *Agency Distance* and *Consumer Cardinality* to differentiate several contexts in which connectors can be used. This differentiation is important because it is not the nature of a connector alone that determines how difficult it is to change a component on which other components depend. In fact, it is the balance between the coupling created by a connector on the one hand and the actual circumstances of how it is being used on

Table 6.5 Connector usage

	Same	Near	Far
1	1-same	1-near	–
n	n-same	n-near	–
N	–	N-near	N-far

the other that determines how a given exposed connector impacts the changeability of a component.

To classify different contexts of how connectors can be used we combine the notions of Agency Distance and Consumer Cardinality. This combination results in the six Connector Usage patterns shown in Table 6.5.

The combinations 1-far, n-far, and N-same are not included because they do not exist in reality. For example, by definition you cannot have an unknown, uncontrollable number of consumers within a single agency boundary.

Table 6.6 illustrates the different connector usages with examples.

If the social or business circumstances of an integration scenario make coordination particularly difficult the connector usages *n-near* and *N-near* should be treated as *n-far* and *N-far,* respectively.

Connector Suitability

The goal of the previous sections was to establish the foundation for quantifying the impact of a given "connection" on how easily and quickly the providing component can be changed.

We have examined two aspects of "connections":

1. The potential change impact on the consumer created by the nature of the connector used.
2. The agency distance and consumer cardinality of the actual use of the connector.

In the following I propose an approach to determine a connector suitability value based on cascading sets of rules that compare the potential change impact of a connector with the circumstances of its use and select one value from a set of connector suitability values.

Connector Suitability Values

A connector suitability value expresses how appropriate the choice of the given connector is compared to the context in which it is used. The following table describes the four suitability values (Table 6.7) used in this chapter.

I have chosen a set of only four values because it is hardly possible to exactly determine connector suitability at a finer granularity. I have also chosen not to define

Table 6.6 Example connector usages

Connector usage	Description	Example
1-same	Exactly one consumer. No expectation or intention that there will be other consumers. Provider and consumer reside within the same agency boundary.	Content Management System and its own database. Two systems connected to synchronize a certain kind of asset.
1-near	Exactly one consumer located in a closely related agency boundary, for example, a subsidiary or a special business partner. Communication serves a special need and is not a general offering of one party. There is no intention to provide the connector to other consumers.	Human Resource department providing data, for example, about open positions, to be included on the corporate Web site.
n-same	Very few consumers; all located within the same agency boundary.	Integration between systems that exist for a single common purpose, for example, Content Management.
n-near	Very few consumers located in closely related agency boundaries. Typically between organizational units of a single enterprise or between few special, selected business partners. There might be an intention to reuse the provided services for other contexts, but the goal is not to offer the service as a product.	Systems integration between organizational units inside a single enterprise or between subsidiaries. Systems integration between business partners to help conducting business (not services sold to partners).
N-near	Very many consumers but located in related agency boundaries. Though coordination is possible because consumers tend to be related through contracts, for example, subscriptions it is highly undesirable for practical and social reasons. Best to be proactively treated as N-far.	Supply-chain-management related service, consumed by many business partners. Online payment gateway offered as a service.
N-far	Very many (potentially millions) or consumers residing in highly unrelated administrative domains. Coordination is impossible.	Online retailer and its customers.

an odd number of values in order to avoid a medium value that would act as a meaningless “catch-all”-value for undecided cases. An even number of possibilities enforces explicit reasoning.

If you find that in your context a higher granularity can be meaningfully supported the methodology can be adapted accordingly by changing the rule sets below. However, to enable comparison between systems or to track quality improvements over time the particular value set and the rules you define are required to remain stable.

Table 6.7 Connector suitability values

Name	Symbol	Definition
Perfect	++	The connector matches the circumstances of its use. There is no possibility to improve the evolvability of the providing component by using a different connector.
Reasonable	+	The connector does not harm the evolvability of the component. It is a good match but if resources permit the system would benefit from changing to a perfectly matching connector.
Problematic	−	The connector does not match the circumstances of its use, but there is also no immediate risk that the mismatch might prevent any kind of system evolution. A necessary change can be difficult and costly but it will be possible to achieve. If resources are available and no critical suitability exists the integration quality of the system will benefit from reducing the number of connectors with problematic suitability.
Critical	–	The connector absolutely does not fit the circumstances of its use. The amount of mismatch can make necessary business-level evolution impossible (for example, rolling out a new product). Critical connector suitability conditions should be improved as soon as resources permit.

Assigning Suitability Values

The proposed methodology uses cascading sets of rules to determine the suitability value of a connector in a given context of use. The advantage of an approach that uses subsequent application of rules is that it enables a refinement of the suitability value selection at different conceptual levels.

The value selection at the first level is based on the architectural analysis of the various connectors without considering any particularities of connector implementation technologies.

The second level allows for refinement of the general suitability values to reflect implementation technology specific properties. For example, Java RMI and Web Services, both Remote Procedure Call connectors, have identical general suitability values but differ significantly when taking the technical properties into account. While Java RMI requires all components to be implemented in the same programming language environment Web services provide language independence. Using the former potentially leads to a programming language change impact while the latter does not. Implementation technology suitability rules can reflect these differences by refining suitability values accordingly.

At the third level it is possible to further adjust the suitability values based on specifics of the application domain or to reflect skills and preferences pertaining to the context or department in which the methodology is used.

After subsequent application of the various cascading levels a suitability value is obtained that can be used to compare different components or to analyze a certain component over time.

General Suitability Value Rules for 1-same

The general suitability of a connector for *1-same* connector usage is determined according to the following rules:

- No connector is *critical* because even broad changes to the single one consumer do not lead to a situation that can make business level requirements impossible to realize.
- Connectors that do not prevent against communication model changes are *reasonable* to indicate that there is a possibility of improvement.
- All other connectors are *perfect.*

General Suitability Value Rules for n-same

The general suitability of a connector for *n-same* connector usage is determined according to the following rules:

- No connector is *critical* because within the same agency boundary and a reasonably small number of consumers it will be possible to handle any given change impact. Even if that implies a complete re-implementation of consumers.
- A connector that has the potential impact of changing the communication model is *problematic*. While it is not immediately critical, this situation is a potential problem and a more suitable connector should be used. Especially if the number of consumers grows or if the agency distance changes, for example, in the course of a company merger.
- Connectors that focus on document-oriented communication, aim for temporal decoupling, and have uniform method semantics are a *perfect* match.
- More tightly coupling connectors are *reasonable* but should be replaced. Especially if the number of consumers is expected to grow.

General Suitability Value Rules for 1-near

The general suitability rules for *1-near* connector usage are the same as those for *1-same* connector usage because the fact that there is only a single potentially impacted consumer makes all kinds of changes achievable with reasonable cost and coordination complexity.

General Suitability Value Rules for n-near

The general suitability of a connector for *n-near* connector usage is determined according to the following rules:

- Connectors that have an impact potential of changing the communication model or the connector protocol are *critical*. Change coordination activity not only involves several owners of consumers but also has to be established across agency boundaries. This combination can make the required changes effectively impossible.
- HTTP Type I is a *problematic* (but not critical) connector. Data format and shared identity oriented changes are costly, but still manageable in an *n-near* context.
- *HTTP Type I*, *Message Oriented Middleware* and *Event-Based Integration* can be considered *reasonable* if they are used at all in this context.
- REST and HTTP Type II are *perfect* connectors.

General Suitability Value Rules for N-near

The general suitability of a connector for *N-near* connector usage is determined according to the following rules:

- *REST* and *HTTP Type II* are *perfect* connectors. The change impact potential of an *HTTP Type II* connector can usually be controlled to only lead to changes that can be coordinated with the customers of a service. Google's HTTP-exposed services are a good example of this.
- *Message Oriented Middleware*, *Event-Based Integration*, and *HTTP Type I* are *problematic* (but not critical) connectors. Data format and shared identity oriented changes are costly, but still manageable in an *N-near* context.
- All other connectors are *critical* because their change impact potential is too large for typical B2C or large-scale B2B integration contexts. It is likely either not possible to achieve the necessary coordination or not desirable to ask the *N-near* consumers to consider it. Also, the potentially large number of consumers requires the impact to be easy to document and relatively easy to adapt to. Both can be achieved with *Message Oriented Middleware*, *Event Based Integration*, and *HTTP Type I* but not with *Remote Procedure Call*, *Shared Database* or *File Transfer*.

General Suitability Value Rules for N-far

The general suitability of a connector for *N-far* connector usage is determined according to the following rules:

- *REST* is the *perfect* connector for this connector use.
- All other connectors are *critical* because it is impossible to achieve coordination with consumers in an *N-far* connector usage context.

Table 6.8 Connector suitability

	1-same	n-same	1-near	n-near	N-near	N-far
File transfer	+	−	+	–	–	–
Shared database	+	−	+	–	–	–
Remote procedure call	++	+	++	–	–	–
HTTP Type I	++	+	++	+	−	–
Message-oriented middleware	++	++	++	+	−	–
Event-based integration	++	++	++	+	−	–
HTTP Type II	++	++	++	++	++	–
REST	++	++	++	++	++	++

General Connector Suitability Summary

The following table summarizes the connector suitability values determined by applying the general suitability rules stated above (Table 6.8).

Integration in a 1-same context is not problematic. Regarding the overall ability to change the system the specific connector used for such point-to-point integrations does not have much influence. However, this situation immediately changes if the number of consumers grows. In this case, system owners should be advised to change the connector before increasing the number of consumers.

On the contrary connector uses that are either public facing or address large-scale B2B scenarios require *REST* or at least *HTTP-Type II* connectors to avoid obstruction of the evolvability of the system.

Connectors that emphasize the exchange of document-oriented messages (as opposed to procedure calls) and aim for temporal decoupling are suitable even for integration with many, not directly controllable consumers.

The connectors with the best suitability, especially in connector usage contexts found in large enterprise IT scenarios, are *Message Oriented Middleware*, *Event-Based Integration*, *HTTP Type II*, and *REST* connectors. They especially support growing numbers of consumers – a property specifically not shared with Remote Procedure Call connectors.

It is important to note that the suitability values and the conclusions drawn from the analysis refer to system evolvability only. They are not intended to express how well a given connector matches the interaction requirements of the components that form a given application. Performance- or real-time considerations, for example, can mandate the use of Event-Based integration connectors although the suitability rules provided yield a negative suitability value. [On the other hand, it is questionable whether publish/subscribe interactions can be economically sustained in *N-near* or *N-far* scenarios (Fielding 2008)].

Table 6.9 Technology specific suitability of RPC connectors

	1-same	1-near	n-same	n-near	N-near	N-far
Remote procedure call	++	+	++	–	–	–
Java RMI	++	+	–	–	–	–
DCOM	++	+	–	–	–	–
CORBA	++	+	++	–	–	–
WS-*(RPC)	++	+	++	–	–	–
RPC-URI tunneling	++	+	–	–	–	–

Additional Rules for Connector Technologies

In addition to the implementation technology, independent rules above further rules can be applied to capture implementation technology properties. The rules presented below are provided as examples and should be extended to cover the properties of the actual technologies used.

Suitability Rules for Remote Procedure Call Connector Technologies

- Some implementation technologies of Remote Procedure Call connectors require the same programming language or programming language environment for the communicating components. These connector technologies are *problematic* when used with more than one consumer because the risk of having to change the implementation of several consumers is undesirable. Examples are RMI and DCOM.
- RPC URI Tunneling is *critical* for any use that involves more than one consumer due to the architectural complexity it involves (for example, the use of idempotent methods to tunnel non-idempotent operations) (Table 6.9).

Suitability Rules for Message Oriented Middleware Connector Technologies

- Some implementation technologies of Message Oriented Middleware connectors require the same programming language or programming language environment for the communicating components. These connector technologies are *problematic* when used with more than one consumer because the risk of having to change the implementation of several consumers is undesirable. Examples are JMS and MSMQ.
- SMTP is a *perfect* connector for large scale messaging (provided the properties of SMTP, for example, its high latency, fit your other architectural needs). The combination of SMTP with other internet standards such as MIME headers and media types are an effective way to address the integration problems resulting from *N-near* and *N-far* usage contexts (Table 6.10).

Table 6.10 Technology specific suitability for MOM connectors

	1-same	1-near	n-same	n-near	N-near	N-far
Message-oriented middleware	++	++	++	+	−	–
JMS	++	++	−	−	−	–
MSMQ	++	++	−	−	−	–
WS-* (Doc-Style)	++	++	++	+	−	–
SMTP	++	++	++	++	++	+

Table 6.11 Technology specific suitability for EBI connectors

	1-same	1-near	n-same	n-near	N-near	N-far
Message-oriented middleware	++	++	++	+	−	–
JMS	++	++	−	−	−	–
WS-Eventing	++	++	++	+	−	–
XMPP Pub-Sub	++	++	++	+	−	–
PubSubHubbub	++	++	++	++	++	+

Suitability Rules for Event-Based Integration Connector Technologies

- Some implementation technologies of Event-Based Integration connectors require the same programming language or programming language environment for the communicating components. These connector technologies are *problematic* when used with more than one consumer because the risk of having to change the implementation of several consumers is undesirable. An example is JMS.
- The PubSubHubbub implementation of Event-Based Integration connectors combines a publish-subscribe protocol with elements of REST, primarily self-describing messages and uniform interface semantics. It relies on open standards and therefore reduces the potential change impact on consumers. It is a *perfect* match for all usage contexts except for *N-far*. PubSubHubbub constrains notification server behavior in a way that can be problematic in an *N-far* environment (Algermissen 2009) thus it is only **reasonable** for these kinds of uses (Table 6.11).

Additional Rules for Local Suitability Tuning

When the proposed methodology is applied in a specific IT environment it might be desired to further tune the suitability values to reflect domain- or business specific nuances. Such a fine-tuning might, for example, be used to express certain technological capabilities of the associated IT departments or aspects of service level agreements with business partners.

Further sets of rules to be applied subsequent to the general and technology specific suitability rules enable such adjustments if necessary.

Component Change Resistance

Components can expose several different connectors in different connector usage scenarios. For example, a monitoring component might expose an *Event-Based Integration* connector to efficiently distribute monitoring events. At the same time it can also expose a *REST*-based API to enable other components to access monitoring reports or configuration settings (Fielding et al. 2010).

Each of the exposed connectors affects how difficult it is to change the component. Assuming that both connectors in the example are *perfectly* matching their context of use the component would exhibit maximum changeability. Its resistance to change would be minimized.

If, however, the component would also expose a *File Transfer* connector to integrate with several systems in two IT departments of a recently acquired company (an *n-near* connector usage) the component's resistance to being changed would be dramatically higher. Given that the component's overall suitability cannot be better than the worst suitability value of all exposed connectors the resulting component suitability value is the minimum value of all exposed connectors.

In the example case, the additional connector would lower the component's value from *perfect* to *critical* (the value of *File Transfer* for the *n-near* context).

Decreasing component change resistance is a primary goal of integration architecture management because it is a precondition for reducing the amount of time necessary to add new functionality to a system.

Integration Architecture Quality

IT systems exist to support business level processes. Associated use cases are realized as networked applications composed of communicating software components.

Integration architecture management aims to optimize the suitability of the applied connectors to maintain an evolvable system. The key property of interest of an integration architecture is how efficiently the integrated system can be changed to respond to new business requirements.

The notion of *component change resistance* described in the previous section can be used to capture this essential IT system property as a measurable entity. The following table assigns evenly distributed percentage values to the four connector suitability values with 100% indicating a *perfect* match (Table 6.12).

Table 6.12 Perfect-suitability percentage

Suitability value	Match percentage (%)
Perfect	100
Reasonable	66
Problematic	33
Critical	0

Using these percentage values we can express the overall suitability of the connectors exposed by a component as a fraction of the desired optimum of a perfect match (100%). For example, the component mentioned in the previous chapter had an original suitability of 100%. When we added the File Transfer connector to integrate with several near-distance consumers the suitability of the component dropped to 0% (critical).

The mapping of suitability values to percentages makes it possible to calculate the average suitability of a set of components. For example, if we have a system consisting of five components with the individual suitability values of 100%, 66%, 66%, 0%, and 100% the average suitability can be expressed as $332/5 = 66.\ \%$. This percentage roughly maps to an average *reasonable* quality of the integration architecture.

The better the individual components score the more adaptable the overall IT system is to requirements for new functionality needed by its stakeholders.

Conclusion

Communication between components causes coupling and this coupling acts as an obstacle against change. This effect needs to be controlled to maintain the ability of an IT system to realize new requirements.

Two forces affect coupling: the nature of the connectors that mediate communication and the number and “distance” of consumers. Choosing connectors that match the circumstances of their use is an essential means to maintain a competitive ability to realize new requirements. Extensive use of mismatched connectors can cause the reactivity of an IT system to stall, leading to critical impact on business evolution.

Most of the problems commonly experienced with integration scenarios are the result of a mismatch between the nature of a given connector and the circumstances of its use.

References

Algermissen J (2009) Message #21263 on atom-syntax mailing list. http://www.imc.org/atom-syntax/mail-archive/msg21263.html. Accessed November 2010

Algermissen J (2010a) Classification of HTTP-based APIs. http://www.nordsc.com/ext/classification_of_http_based_apis.html#uri-rpc. Accessed November 2010

Algermissen J (2010b) Classification of HTTP-based APIs. http://www.nordsc.com/ext/classification_of_http_based_apis.html#http-type-one. Accessed November 2010

Algermissen J (2010c) Classification of HTTP-based APIs. http://www.nordsc.com/ext/classification_of_http_based_apis.html#http-type-two. Accessed November 2010

Baker M (2005) Towards truly document oriented Web services. http://www.coactus.com/blog/2005/07/towards-truly-document-oriented-web-services/. Accessed November 2010

Birrell AD, Nelson BJ (1984) Implementing remote procedure call. *ACM Transactions on Computer Systems*, 2, 1984, pp. 39–59

Bosak J, McGrath T (2006) Universal business language 2.0. OASIS. http://docs.oasis-open.org/ubl/cs-UBL-2.0/UBL-2.0.html. Accessed November 2010

Box D et al. (2006) Web services eventing (WS-Eventing). W3C. http://www.w3.org/Submission/WS-Eventing/. Accessed November 2010

Fielding RT (2000) Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine

Fielding RT (2008) Economies of scale. Fielding, R.T. http://roy.gbiv.com/untangled/2008/economies-of-scale. Accessed November 2010

Fielding RT (2010) Fielding, Roy Thomas, Message #15819 on rest-discuss mailing list. http://tech.groups.yahoo.com/group/rest-discuss/message/15819. Accessed November 2010

Fielding RT, Gettys J, Mogul JC, Nielsen HF, Masinter L, Leach P, Berners-Lee T (1999) Hypertext Transfer Protocol – HTTP/1.1. *Internet RFC 2616*

Fitzpatrick B, Slatkin B (2010) PubSubHubbub Core 0.3 – Working Draft. Google Inc. http://code.google.com/apis/pubsubhubbub/. Accessed November 2010

Google (2010) Google calendar API. http://code.google.com/apis/calendar. Accessed November 2010

Hadley M (2009) Web application description language. http://www.w3.org/Submission/wadl. Accessed November 2010

Hophe G, Woolf B (2004a) Enterprise Integration Patterns. Pearson Education. p. 43

Hophe G, Woolf B (2004b) Enterprise Integration Patterns. Pearson Education. p. 47

Khare R, Taylor RN (2004) Extending the Representational State Transfer (REST) Architectural Style for Decentralized Systems, in 26th International Conference on Software Engineering (ICSE), (Edinburgh, Scotland, 23–28 May 2004)

Klensin J (2001) Simple mail transfer protocol. http://www.ietf.org/rfc/rfc2821.txt. Accessed November 2010

Microsoft (2007) Distributed component object model. Microsoft. http://msdn.microsoft.com/library/cc201989.aspx. Accessed November 2010

Microsoft (2009) Message queuing MSMQ. Microsoft. http://msdn.microsoft.com/en-us/library/ms711472(VS.85).aspx. Accessed November 2010

Millard P, Saint-Andre P, Meijer R (1999) XEP-0060 Publish-Subscribe. XMPP Standards Foundation. http://xmpp.org/extensions/xep-0060.html. Accessed November 2010

OMG (2008) Common object request broker architecture. Object Management Group (OMG). http://www.corba.org/. Accessed November 2010

Oracle (2009) Java remote method invocation. http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html. Accessed November 2010

Shaw M, Garlan D (1996) Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, Englewood Cliffs, NJ, USA. p. 169

Sun (2008) Java Message Service (JMS). http://www.sun.com/software/products/message_queue/index.xml. Accessed November 2010

Tilkov S (2005) RPC style web services. http://www.innoq.com/blog/st/2005/05/18/rpcstyle_web_services.html. Accessed November 2010

Twitter (2009) The Twitter API. http://apiwiki.twitter.com/Twitter-API-Documentation. Accessed November 2010

Winer D (1999) XML remote procedure calls. http://www.xmlrpc.com/spec. Accessed November 2010

Chapter 7
FOREST: An Interacting Object Web

Duncan Cragg

Abstract FOREST satisfies the need for objects to easily interact across the network in a RESTful way – without calling methods on each other. To do this, it asks you to set your objects up in an Observer Pattern relationship. Or, in particular, a "Functional Observer Pattern", where an object's state is set as a Function of its current state plus the state of other objects it Observes through links. This observation occurs through either pull or push of linked object state. Such a programming model maps directly to RESTful distribution over HTTP, using GET for pull and POST for push of object state, in both directions between interacting servers. Objects are published into a global interacting object Web. This distributed object architecture is declarative in nature, and thus very expressive, as well as being naturally concurrent.

FOREST satisfies the need for objects to easily interact across the network in a RESTful way – without calling methods on each other. To do this, it asks you to set your objects up in an Observer Pattern relationship. Or, in particular, a "Functional Observer Pattern".

FOREST is an acronym for "Functional Observer REST". Once objects are in a Functional Observer relationship, RESTful distribution or integration becomes very simple, as there is essentially a one-to-one mapping from the Functional Observer model to REST's Hypermedia Constraint.

FOREST is thus a distributed object pattern – distinguished by its style of object interaction. It describes a Web of interlinked, interacting, interdependent object resources, hosted across multiple applications or servers.

D. Cragg (✉)
ThoughtWorks (UK) Ltd., Berkshire House, 168–173 High Holborn, London, WC1V 7AA
e-mail: restbook@cilux.org

Functional Observer means that these objects interact by setting their next state as a Function of their current state plus the states of other objects Observed near them in the Web and on which they depend. Observation may occur either by pull or by push of object state.

This observation is enabled across the network by an HTTP layer using just GET and POST, in a "symmetric REST" style, conforming to the constraints of REST (Fielding 2000). A server may become a client in order either to GET or pull a remote object resource for its own dependent object resource, or to POST or push its own object to a remote dependent object. Another way of looking at this is that a client becomes a first-class server to publish its own object resources.

Although an unusual object interaction pattern at the programming language level, the Functional Observer model is actually very easy to program due to its declarative, functional, reactive and asynchronous programming style. Functional Observer objects are programmed independently as "masters of their own evolution". It is similar to approaches found in the Clojure and Erlang programming models, and may be readily implemented over the asynchronous Node.js framework, as well as mainstream languages such as Java.

This object independence makes Functional Observer object interactions easy to distribute, both across application partitions – inheriting the interoperability, evolvability and scalability benefits of REST – and across multicore, without the usual concerns around threads and locks.

Functional Observer Pattern

In the Functional Observer Pattern, an object sets its own next state as a Function of the object states it Observes in itself and through its links at any time (Fig. 7.1):

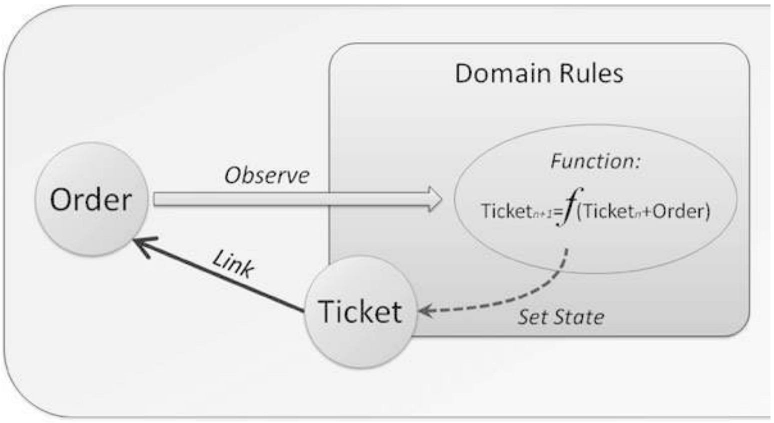


Fig. 7.1 Functional Observer

Functional Observer objects do not interact by calling methods, they simply exchange public state with each other. An object is master of its own evolution. When it needs to set its own state – for example, when observed itself, or after being notified of an update to a linked object it is observing – it looks at its own content state and the state of all other objects around it – local and remote – that are visible through links (or links to links via a chain of objects) and on which it depends. Then it moves its own state forward, guided by functions describing the application business rules or domain-level constraints. That new state is then notified on in turn to other dependents or observers.

An important constraint in Functional Observer is that observed objects do not get to know what other objects are observing them. However, an object "A" can make itself known to another object "B" that does not yet have any link access to it, by pushing itself to "B" unasked. Object "B" can then set itself up as an observer of object "A" if it then adds a link to "A" and carries on scanning it.

Objects have unique ids, perhaps UUIDs or GUIDs, by which they are fetchable or observable from an object cache. Their content – the data that has a state at any time – is a structure of common programming language elements: strings, numbers, lists and hashes, with links to other objects via their unique ids.

Functional Observer and Related Styles

It is beyond the scope of this chapter to discuss the history and origins of the Functional Observer model and its place within the broad scope of programming models, patterns, styles and paradigms. We will simply point out some similarities and mention some languages that support implementation.

The Functional Observer model is somewhat related to Dataflow, Reactive and Functional Programming styles, although it can be readily implemented in mainstream languages. It has some similarities to, and perhaps may be implemented by, Clojure's Agents and Watchers (Hickey 2009). Similarly, implementations of the Actor Model, including Erlang (Armstrong et al. 1996), may support the Functional Observer Pattern. A useful, but more basic, infrastructure for implementation is provided by the "Node.js" asynchronous, server-side Javascript framework (Dahl 2009).

Functional Observer objects interact in a declarative, reactive and asynchronous way with other objects. A Functional Observer object *never directly tells another what to do* by calling methods on it. (It may strongly imply a preference directed at another, however, which can have a similar effect, but most often that style of interaction is not needed.) This switch from imperative to declarative thereby causes a corresponding inversion of the Object-Oriented mantra, "Tell don't Ask" (Cragg 2009).

Implementation of Functional Observer

The implementation of Functional Observer is very simple. Normally, a callback on an object will be set up, triggered by a pushed incoming state change on a previously observed, linked object; or by object creation or re-cacheing. Then the object will go around scanning links to observe, and pulling state on which it depends, to decide what its current state should be, or what its reaction will be to an incoming new state. When it changes itself, its state gets pushed on in turn to the objects that have previously depended on it.

The act of scanning an object through a link (or link chain) sets up the observation of that object's next update. Observation can occur either by pull – the observer reads the state of the linked target object when needed – or push – the target pushes its state to the observer when it changes. This can thus support both lazy and eager programming models.

An object observes the links it visits in this process, and conversely, if it no longer visits a link, it stops observing it. An object that falls out of the cache through lack of use may also stem observation of its dependencies. Some objects may allow themselves to be observed at any time, some only when they have re-evaluated themselves by observing their own dependencies.

A difference between Functional Observer and the traditional Observer Pattern is in the way events are notified. Functional Observer operates asynchronously, rather than pushing out the notifications on a single thread. Hence notifications are queued, and thus may be handled by another thread or another process on a remote machine. Reads – cacheing objects in and GETs of remote objects – are also asynchronous. This asynchronicity means that an object needs the option of deciding to block or refuse observers when its state is not ready; when it is waiting for a dependent state of its own to ensure it is up-to-date.

Functional Observer REST

The familiar model of method-calling that is used for object interactions in common programming languages is, of course, not resonant with distribution in the REST style. We identify this method-invocation interaction style as "imperative". RPC is an example of this style when attempted over the network.

Functional Observer's declarative object interaction style, on the other hand, *is* resonant with REST, and maps straightforwardly onto RESTful use of HTTP; and in particular onto the Hypermedia Constraint, as we will show. Correct REST distribution or integration has often been seen as a tricky art, so there is a huge advantage in having a simple and powerful programming model where RESTfulness almost "drops out" (Fig. 7.2).

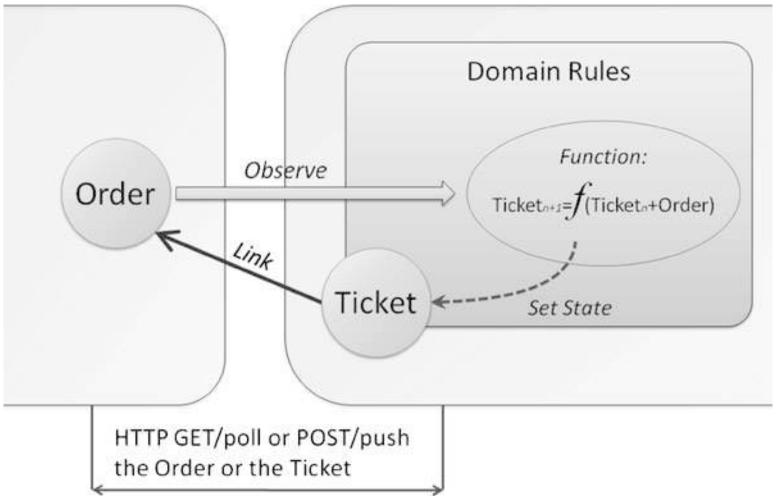


Fig. 7.2 Functional Observer REST

FOREST's Functional Observer object interactions map onto HTTP bidirectionally: to pull linked object resource state using GET, and to push state using POST, in the style of a "reverse GET". As a result, using POST this way is an idempotent operation. Object content is serialised into an appropriate Media Type.

POST notifications occur by the sender knowingly pushing itself to a target object: observation and awareness of updates is manifest onto HTTP as either conditional GET and polling or POST for this deliberate push. As objects do not get to know their observers, a GET need not carry information about which object is asking.

These state transfers by GET or POST are all that is needed to allow each object resource to meet its domain logic dependencies on other linked objects. HTTP is only used, in this push–pull mode, for the domain-independent exchange of state over the network; all domain-level conversations occur only up at the level of the object state itself, within the Functional Observer Pattern interactions. Using the HTTP layer for generic state transfer, supporting the interactions in the object layer, gives clear separation of concerns.

One consequence of Functional Observer's declarative interaction style mapped into HTTP in this way is that the two, less-used, HTTP verbs that often come up in REST patterns – PUT and DELETE – have no place. There is nowhere in FOREST where it makes sense to imperatively or directly attempt to replace or delete *someone else's object*. HTTP is not used by a client application to try and tell a server application what to do; it is just used for the bi-directional state exchange by pull and push, GET and POST.

It is via this network distribution that we move from just Functional Observer to FOREST, which as we will show, naturally follows the constraints of REST.

FOREST Foreign Exchange Trading Example

This will all become much clearer with a real example. Rather than dig into how a programmer sees the Functional Observer Pattern play out, with all the details of implementation languages and programming styles, it will be enough simply to watch what happens "on the wire" in HTTP messages, as some objects held on different servers interact with one another.

This will bring out the detail of both the Functional Observer interactions and the application of the constraints of REST in FOREST. The second part of this chapter will then show in detail how FOREST meets the Hypermedia Constraint, which is a little more complex than can be explained by way of this example. The Stateless Constraint is also left to be discussed in that context.

We'll demonstrate with a Foreign Exchange order fulfilment scenario. This will have an Order server holding Orders and a Fulfilment server holding fulfilment status as Tickets. The Order server can also hold Payments. Hence, in the Functional Observer style required by FOREST, Orders may observe Tickets and Tickets may observe Orders and Payments.

First, an as-yet-unseen Order in the Order server (fx-orders.com) observes, and thereby fetches, a known, top-level "Dealer" object from the Fulfilment server (fx-broker.com):

```
GET /dealer123 HTTP/1.1
Host: fx-broker.com

HTTP/1.1 200 OK
Etag: "313"
Cache-Control: max-age=10
Content-Type: application/json

{ "tags": [ "fx", "dealer" ],
  "tickets": [
               "http://fx-broker.com/tick110",
               "http://fx-broker.com/tick109"
             ]
}
```

Just this short, simple interaction gives us very much to discuss about the application of REST's constraints in FOREST.

Client–Server; Layered; Cache

So: which is the client and which is the server? In this scenario, as we will see, the Order server can be a client of the Fulfilment server and vice-versa.

In general, servers have responsibilities at a domain or business level, which are not always easy to partition into simply either client and server at the network level.

But if we look over Fielding's thesis, we can see that "separation of concerns is the principle behind the client–server constraints" and "the separation allows the components to evolve independently, thus supporting the Internet-scale requirement of multiple organizational domains."

Thus having servers that are also clients is fully compatible with the intent of the Client–Server Constraint as long as this separation of concerns exists in our Order and Fulfilment servers. Also, using HTTP means the client–server interaction style is baked in at the level below.

Finally, the REST definition states as an advantage of the uniform interface that servers such as search engine crawlers can act as clients. We will refer back to this below, as it is one of the closest ways the Web comes to our fulfilment scenario.

We'd like to be able to use proxy-caches and other intermediaries, via the Layered Constraint, which we can do by using HTTP correctly. It actually ties in nicely to our more two-way version of the Client–Server Constraint, since proxies are also both clients and servers. Further, our client-like servers can act as intermediaries when isolating concerns in more complex scenarios than this fulfilment one.

We have some headers that will help us meet the Cache Constraint of REST, in the Etag and max-age. We should be able to cache Orders in the Fulfilment server, and Tickets in the Order server, as well as in any proxy-caches in between.

Identification

In FOREST, we only have real object resources of the host's implementation language, not resources that are in any way abstract. They will have unique ids in that internal object world. Hence, they can always respond to GET requests on URLs that map to their ids with their object representations or serialisations.

Now, we've given the Dealer URL a simple, readable name. But in real life, that URL is more likely to look something like: "http://fx-broker.com/xyz/312a–5990bf4–34cd007.json" – with an internal GUID or UUID appearing in the URL.

Notice that we've used the more common acronym "URL" here, instead of the more generic or "correct" one, "URI", that is most often used in REST discussions. The definition of "URI" includes potentially unfetchable referents, identified by strings intended for human consumption, such as URNs [or URLs with fragment identifiers (Berners-Lee et al. 2010)]. Our URLs are always fetchable and always opaque, at least in principle. "URL" has a clear meaning that is exactly what we want (Mealling and Denenberg 2002; Berners-Lee et al. 2005) and which allows us to be very precise when discussing the REST Identification Constraint, thus avoiding any philosophical or Semantic Web entanglements. Anything semantic has to appear in an object's content, *not* in the links between objects.

If including an object UUID or GUID in its URL, the "Identifier" aspect of the URL is essentially given by that element. Then the domain or host part of the URL is the "locator" aspect for where that uniquely- and universally-identified object can always be found. Indeed, any other characters of a URL are then essentially redundant, including ".json" or ".xml", other path elements, etc.

Further, given that the UUID or GUID is universally or globally unique, it is certain that a representation or copy of the object referred to could also be found cached in any one of a number of other hosts – perhaps keyed by the unique object id alone. It is the ability to reliably find or locate an object representation by a unique identifier that is the essential characteristic we need.

Self-Descriptive Media Types

We are using JSON for our serialisation of objects into their representations – the Content-Type is application/json – as it is simple and clear, and a better match for the kind of data we prefer to program our objects with.

This also allows us to conform to REST's Self-Descriptive Constraint for Media Types as this is a common, standardised format, widely understood by available libraries and frameworks.

The purpose of the Content-Type header at the HTTP layer is to give broad direction to choose processing modules within which most of the actual content interpretation happens, once that module starts reading inside the content itself. In the content layer, there will always be an understanding between client and server that goes beyond the Media Type and into business or domain interactions.

For example, there is a channel of communication set up between an author of plum jam recipes and their plum-picking readers, carried through HTML. On an e-commerce Web site selling plum jam, HTML is exchanged such that the site can have an e-commerce interaction with the jam-consuming end-user. That exchange is carried by, or executed through, the base HTML Media Type.

There is not one Media Type for recipes, and another for buying jam! Minting new Media Types, maybe one or more Media Types per application – perhaps as "vnd.*" or "*+xml" types – compromises self-descriptiveness.

At the HTTP level on the Web, the Content-Type is a very broad and coarse label, which has the great advantage of self-descriptiveness: REST requires that we have few Media Types understood as widely as possible; including by installed libraries and applications. Our guidance on Media Types is to follow the crowd – even, indeed, in preference to following a standard.

So we should use a very common base type, called out in HTTP's Content-Type, and have all the rich domain-level semantics up inside the content, driving the Functional Observer interactions. Hence, we could re-use XML or XML-based types such as XHTML1 or XHTML5 or Atom and AtomPub to serialise objects. More appropriately and simply for our data-oriented applications, we could output JSON representations of the internal object resources. Or we could use content negotiation to choose the serialisation.

But JSON is very bare and basic compared to XHTML. As we gain experience, we could aim to settle on a single, common, standardised sub-Media Type of JSON, with various domain-specific formats or schemas inside that type – such as our Orders, Payments and Tickets. Perhaps we may call it "application/forest+json", following the "+xml" convention.

All this depends to some extent on how you view the strictness of the Self-Descriptive Constraint with respect to Media Types, which Fielding has called out as something of a frustratingly variable constraint in REST (Fielding 2006).

Self-Descriptiveness Inside the Content

As a publisher, unless you make a reasonable attempt to re-use all or part of any formats or schemas within XML, XHTML or JSON that are in wide use or being standardised, including Microformats, etc., you are still reducing your conformance to REST's Self-Descriptiveness Constraint. You still need to be aware of the trade-off of precision versus re-use that this implies.

As long as the base Media Type is conformed to, this re-use can be in the form of basic instantiation, sub-classing or extending. Also, the base type can be used as a carrier for another type, in the way Microformats, RDFa or Microdata are carried by XHTML1/5 and content is carried by Atom. It may also take the form of just re-using parts of existing standards.

On the client side, however, different recipients of such data see things their own way. Class is in the eye of the beholder, not synchronised through shared libraries. This allows decoupling and independent evolution.

The way a Media Type is interpreted or processed is entirely up to the client. The server is signalling what it thinks this content is in general terms, but the client can and will interpret that Media Type however it likes given its current state and goals. Whenever you fetch a JSON- or an XML-based object, the schema above that base – i.e., where you expect to find the data you were looking for – will always depend on the URL you fetched it from, the context around that link, and the use you expect to make of it.

In the spirit of Postel, decoupling is enhanced through a server's respect of standards in its published data, along with a client's tolerance of the incoming type – just taking what it understands and ignoring anything else, perhaps by using XPath or an equivalent approach in JSON.

In our JSON content, we've chosen a tagging system for creating class-like subtypes of the base Media Type. The "fx" and "dealer" tags give us flexible, loose typing which sets our expectations about how we interact with this object, from reading some documentation and conformance tests. The aim would be to settle on a number of common JSON schemas or grammars, using such conventions, within our "application/forest+json" Media Type.

Self-Descriptive Methods, Headers and Response Codes

REST does not talk much about the subject of methods, except to say that they should form a uniform interface and that they should be well-known.

As we've already described, our use of GET and POST is entirely for state transfer in each direction – initiated by either side. This is pretty conformant to Self-Descriptiveness, as the Web works much like that. The idempotency of POST in FOREST can be signalled by use of our "application/forest+json" Media Type.

Using PUT and DELETE can actually fail us in self-description of messages, since much HTTP code in the wild does not handle them. However, of course, they are not necessary for the Functional Observer Pattern.

Correct use of common HTTP headers and response codes gives proxies and clients a chance to cache content and to know when things have gone wrong in the state transfer. We expect to see the usual response codes: "200 OK", "303 See Other", "304 Not Modified", "404 Not Found", and others.

Further helping self-descriptiveness and separation of concerns, these headers and return codes should be given as closely as possible their common meanings regarding just the state transfer of representations. In FOREST, we do not modify their meaning in any way on a domain-specific basis. Thus "200 OK" means the state transfer was successful – even if that state indicates an "error mode" at *domain* level, in a Functional Observer object exchange.

Hypermedia and Hyperdata

Although we will revisit the Hypermedia Constraint later, we still need some hypermedia to constrain! In fact, in FOREST, our hypermedia is more in the nature of "hyperdata". Hyperdata means having our data linked up into an object Web or a graph of objects.

Here, in the response content representing the Dealer object, we see that the JSON object returned has URLs inside JSON strings. It is currently dealing with a number of Tickets, listed within it.

These links are created in a fairly obvious way. Presumably our Dealer object links internally to these Ticket objects. So when serialised into a Dealer representation, those links can simply be converted to URLs containing the Ticket objects' unique ids. As we'll see below, the Ticket object will also point by URL to its corresponding Order object over on the Order server – which points back – thus completing a cross-server, hyperdata object Web.

Now, declaring that a URL found in a JSON string is hyperdata is hardly something worthy of an RFC, but that would be part of any "application/forest+json" JSON hyperdata standard. Similarly, you may prefer XHTML1/5 to XML, as it comes with a built-in link semantics, where XML would need to be enhanced with XLink.

Links in FOREST *only* represent such declarative data structuring, and therefore always point to fetchable object resources. Link relations may be used in XHTML or XML as part of the serialisation of the structure of an object linked to another, but the entire surrounding context of a link, plus the goals and intents of the observer, give the link semantics, not just a single relation tag. Finally, a link may sometimes just have to be fetched before you can really know what it meant!

As long as all of your interacting servers are able to discover and use the links they need that are in the data, guided by surrounding context, that is all that matters. Thus, we can see and traverse links in our hyperdata. We have a distributed graph of objects; a hyperdata object Web, both created and consumed by the applications being integrated.

Back to the Example…

After that very long detour, triggered by a simple GET, things should go a little quicker now in our Foreign Exchange example. Here is that GET by the Order observing the Dealer again:

```
GET /dealer123 HTTP/1.1
Host: fx-broker.com

HTTP/1.1 200 OK
Etag: "313"
Cache-Control: max-age=10
Content-Type: application/json

{ "tags": [ "fx", "dealer" ],
  "tickets": [
               "http://fx-broker.com/tick110",
               "http://fx-broker.com/tick109"
              ]
}
```

An Order object is coded to understand Dealer objects. It knows that this is the entry or starting point of its transaction, as long as it can make that Dealer notice. So it sets itself up to trigger a POST of its own state to that Dealer – an unsolicited push or notify in Functional Observer terms, which can make the Dealer an observer of the Order as long as it is interested:

```
POST /dealer123 HTTP/1.1
Host: fx-broker.com
Content-Type: application/json

{ "%url": "http://fx-orders.com/ordr321",
  "tags": [ "fx", "order" ],
  "params": [ "usd/jpy", "buylim", 81.7, 500.00 ],
  "dealer": "http://fx-broker.com/dealer123"
}
```

This JSON Order object includes its own URL at the Order server, with a preceding "%" on the tag as a convention indicating a metadata field. Such fields are candidates for including in the HTTP headers; the URL could be included as a "Content-Location" header, for example. This depends on your tolerance of non-standard HTTP header usage.

The appearance of this URL reflects firstly, that this is a first-class object available on the Order server whose state is being pushed, and secondly, that this POST is idempotent, as are all FOREST POSTs, because reporting the latest or current state is an idempotent operation. This POST is a state declaration; it is idempotent in intent – the Order is just telling the Dealer of its current state – implying that it is ready to be told about a corresponding Ticket. POST is state transfer in FOREST, not an event, message, action or command. The role of POST will be discussed further below.

The "dealer" URL in the Order object indicates that this is an Order for this Dealer, no other. The Order has to stand independently, declaratively; potentially fetched by others using GET. There will be many such links in Functional Observer, since interactions within and through an object Web is fundamentally how it all works.

The order itself is for a USD/JPY exchange – "buylim" is short for "buy limit" which means buy when the price drops far enough – below 81.7 here.

On the Fulfilment server side, there is a constraint or rule on the Dealer that ensures it interprets this incoming POST as a state declaration, not a command that could result in multiple Tickets: "if I observe an Order pointing to me that does not have a link to a Ticket, I must make sure I have a Ticket somewhere in my list that points to this Order". Such rules work off state, not state change, so can be re-run repeatedly.

So, within the Fulfilment server, the Dealer object sees the incoming Order object and creates a new Ticket object, adding that object to its list.

Then it is up to the Ticket to carry on the conversation with the Order. The Ticket object is now a "live" object, responsible for its own evolution. Once the Ticket has sorted itself out, it needs to tell the Order object about itself.

A POST response is something of an open channel for now *returning* state. The HTTP RFC is rather vague about what is returned from a POST. In theory, it would be possible to always return a "204 No Content" empty response to all POSTs, and then POST back any follow-up updates separately. But that would be a waste – it is more efficient to use this opportunity to update the POSTing server immediately with any new state that has occurred as a result of the incoming POST.

So, the most likely response to a POST, and usually the most useful, is for it to act exactly as if it were a GET on the POST URL target. It returns a 200 code and sets the Content-Location header to its own URL, to indicate that this response is intended to be seen as equivalent to a GET (Fielding et al. 2011). This gives the target a chance to immediately and efficiently return in the response any new state that was triggered by the incoming object POST.

Here is such a response that we *could* get to the POST on the Dealer, showing our new Ticket object in its list:

```
HTTP/1.1 200 OK
Content-Location: http://fx-broker.com/dealer123
Etag: "314"
Cache-Control: max-age=10
Content-Type: application/json

{ "tags": [ "fx", "dealer" ],
  "tickets": [
               "http://fx-broker.com/tick111",
               "http://fx-broker.com/tick110",
               "http://fx-broker.com/tick109"
              ]
}
```

But in this case, it is not telling us much. Alternatively, we could return another object that is considered a dependency of that incoming object, either through the same 200/Content-Location approach, or through a 303 redirect, also containing the object itself to save a round-trip. In this example, the Ticket object itself would actually be more useful to us, instead of just the Dealer pointing to it:

```
HTTP/1.1 303 See Other
Location: http://fx-broker.com/tick111
:
```

or:

```
HTTP/1.1 200 OK
Content-Location: http://fx-broker.com/tick111
Etag: "1"
Cache-Control: max-age=10
Content-Type: application/json

{ "tags": [ "fx", "ticket" ],
  "order": "http://fx-orders.com/ordr321",
  "params": [ "usd/jpy", "buylim", 81.7, 500.00 ],
  "ask": 81.9,
  "status": "waiting"
}
```

The Ticket object confirms the params, current asking price and status.

If for some reason the first approach were actually taken, the Ticket would have needed to push itself back at the Order, again by an unsolicited push:

```
POST /ordr321 HTTP/1.1
Host: fx-orders.com

{ "%url": "http://fx-broker.com/tick111",
  "tags": [ "fx", "ticket" ],
  "order": "http://fx-orders.com/ordr321",
  "params": [ "usd/jpy", "buylim", 81.7, 500.00 ],
  "ask": 81.9,
  "status": "waiting"
}

HTTP/1.1 204 No Content
```

[The Cache-Control and Content-Type headers will not be shown from now on, for brevity.]

These would be alternative mappings to the HTTP layer of the Functional Observer layer's Ticket pushing itself to the Order.

As seen here, a POST can also return a "204 No Content" when there is no interesting new return state to report. In theory, a "202 Accepted" could be used instead of 200s or 204s, but that is not particularly meaningful in FOREST, since it is all asynchronous.

Finally, an object can indicate that it is not interested in, or no longer dependent on, a pushed/POSTed object, by returning a "403 Forbidden" status code (or a 405 if it is *never* interested in POSTs). Note that, although this may stem the POSTs at the HTTP level, the pushing object should not depend on these error status codes to decide what to do: it should only look at and depend on the content of the troublesome push target.

All POSTs can be retried if the response was not received, POST being idempotent in FOREST. As can all GETs, of course, including for polling. Notice that max-age gives our polling algorithm something to work on when calculating an optimum polling frequency. If this Ticket state did not get POSTed after a time period, then either the Order could be re-POSTed if the Order server timed out first, or the Ticket could be re-POSTed if the Fulfilment server noticed its POST failed. Note that max-age is set by the object itself. Also note that there will be both HTTP-level and domain- or business-level timeouts and retries.

Using just the two HTTP methods, GET and POST, and their corresponding headers and status codes and appropriate timeouts and retries, to manage the pull and push of object state between interdependent objects, leaves all the business-level or domain-specific object interactions separated up into the layer above – in the content and its types and formats over or within the common Media Type. Which is also how the Web works.

Back to our example: the asking price is currently too high. But we are confident of a fall, so we've decided we'd like to invest more. The Order is updated, and pushes itself now directly at its own Ticket:

```
POST /tick111 HTTP/1.1
Host: fx-broker.com

{ "%url": "http://fx-orders.com/ordr321",
  "tags": [ "fx", "order" ],
  "params": [ "usd/jpy", "buylim", 81.7, 1000.00 ],
  "dealer": "http://fx-broker.com/dealer123",
  "ticket": "http://fx-broker.com/tick111"
}

HTTP/1.1 200 OK
Content-Location: http://fx-broker.com/tick111
Etag: "2"

{ "tags": [ "fx", "ticket" ],
  "order": "http://fx-orders.com/ordr321",
  "params": [ "usd/jpy", "buylim", 81.7, 1000.00 ],
  "ask": 81.8,
  "status": "waiting"
}
```

We are in luck, the market is still heading down. The Ticket is updated accordingly: its state is a function of the params in the order, combined with its internal access to the Foreign Exchange market.

Time passes. Anxiously, we poll to see if we missed anything because of a dropped POST:

```
GET /tick111 HTTP/1.1
Host: fx-broker.com
If-None-Match: "2"

HTTP/1.1 304 Not Modified
Etag: "2"
```

No – nothing is changed, the market is not moving.

Ah – now the price has dropped – and our Order is filled. We immediately get a notification pushed to the Order:

```
POST /ordr321 HTTP/1.1
Host: fx-orders.com

{ "%url": "http://fx-broker.com/tick111",
  "tags": [ "fx", "ticket" ],
  "order": "http://fx-orders.com/ordr321",
  "params": [ "usd/jpy", "buylim", 81.7, 1000.00 ],
  "ask": 81.6,
  "status": "filled"
}

HTTP/1.1 204 No Content
```

Now, while waiting for that we started to feel even more confident in our USD/JPY prediction, and wanted to bet on an even cheaper price. However, we now have a race condition:

```
POST /tick111 HTTP/1.1
Host: fx-broker.com

{ "%url": "http://fx-orders.com/ordr321",
  "tags": [ "fx", "order" ],
  "params": [ "usd/jpy", "buylim", 81.5, 1000.00 ],
  "dealer": "http://fx-broker.com/dealer123",
  "ticket": "http://fx-broker.com/tick111"
}

HTTP/1.1 200 OK
Content-Location: http://fx-broker.com/tick111
Etag: "4"

{ "tags": [ "fx", "ticket" ],
  "order": "http://fx-orders.com/ordr321",
  "params": [ "usd/jpy", "buylim", 81.7, 1000.00 ],
  "ask": 81.6,
  "status": [ "filled" , "not-as-ordered" ]
}
```

Now the params do not match and it is too late, so this is flagged in the JSON "status" field. As explained above, this *domain-level* "error" condition is a Functional Observer object state exchange that still gets *transferred* at the HTTP level with a "200 OK" status.

Temporarily frustrated, we cancel the Order, with an update to its state:

```
POST /tick111 HTTP/1.1
Host: fx-broker.com

{ "%url": "http://fx-orders.com/ordr321",
  "tags": [ "fx", "order" ],
  "params": "cancelled",
  "dealer": "http://fx-broker.com/dealer123",
  "ticket": "http://fx-broker.com/tick111"
}
```

```
HTTP/1.1 200 OK
Content-Location: http://fx-broker.com/tick111
Etag: "5"

{ "tags": [ "fx", "ticket" ],
  "order": "http://fx-orders.com/ordr321",
  "params": [ "usd/jpy", "buylim", 81.7, 1000.00 ],
  "ask": 81.6,
  "status": "cancelled"
}
```

No problem, the Ticket is marked as cancelled – presumably the Dealer thinks that that was indeed a good price and will keep the purchase themselves. Again, we do not use DELETE because cancellation is a domain-level object interaction, and DELETE does not make sense when we are using the HTTP level for state transfer only – what would you delete? Trying to delete anything would be an imperative call, and this is all about state declaration.

Actually, on second thoughts, we'll take it at that price:

```
POST /tick111 HTTP/1.1
Host: fx-broker.com

{ "%url": "http://fx-orders.com/ordr321",
  "tags": [ "fx", "order" ],
  "params": [ "usd/jpy", "buylim", 81.7, 1000.00 ],
  "dealer": "http://fx-broker.com/dealer123",
  "ticket": "http://fx-broker.com/tick111"
}

HTTP/1.1 200 OK
Content-Location: http://fx-broker.com/tick111
Etag: "6"

{ "tags": [ "fx", "ticket" ],
  "order": "http://fx-orders.com/ordr321",
  "params": [ "usd/jpy", "buylim", 81.7, 1000.00 ],
  "ask": 81.6,
  "status": "filled"
}
```

Not too late. Notice how this is all state-driven, not event-driven. All dependencies are state-dependencies. We are just applying business domain rules or constraints over dependent state. And in this case, as long as the back-end systems still allow the order to be filled at the stated conditions, we are on. There is no rigid state machine telling us we cannot go from cancelled back to ordered-and-filled. The business domain rules constrain relative state at any time, not a strict sequence of events.

Now to pay for the deal. We add a link to a new, locally-hosted Payment object:

```
POST /tick111 HTTP/1.1
Host: fx-broker.com

{ "%url": "http://fx-orders.com/ordr321",
  "tags": [ "fx", "order" ],
  "params": [ "usd/jpy", "buylim", 81.7, 1000.00 ],
  "dealer": "http://fx-broker.com/dealer123",
  "ticket": "http://fx-broker.com/tick111",
  "payment": "http://fx-orders.com/paym432"
}

HTTP/1.1 204 No Content
```

Now the Ticket, that can see this new Order state, needs to traverse the “payment” link to see it, to determine its next state – hopefully to “paid”. Hence there is no change just yet to the Ticket, so it returns a 204, then fetches the Payment:

```
GET /paym432 HTTP/1.1
Host: fx-orders.com

HTTP/1.1 200 OK
Etag: "1"

{ "tags": "payment",
  "invoice": "http://fx-broker.com/tick111",
  "order": "http://fx-orders.com/ordr321",
  "amount": 81600.00,
  "account": { .. }
}
```

We could have POSTed this Payment to the Ticket instead, after POSTing the Order pointing at it. It is only shown this way as a reminder that the state transfer is essentially independent of which server initiates it.

If we POSTed, the idempotency of that FOREST state transfer would ensure multiple submissions were ignored, assuming the server co-operates. Again, there is probably a rule in there on Ticket that says: “If I’m aware that there is a Payment object pointing to me and I’m unpaid, take the payment (once) using that Payment object”.

Notice that this can be a generic, non-Foreign Exchange domain, payment format or type – it does not say “fx” in the tags. And we can use normal Web security such as Auth headers, TLS, etc., to make this secure.

```
POST /ordr321 HTTP/1.1
Host: fx-orders.com

{ "%url": "http://fx-broker.com/tick111",
  "tags": [ "fx", "ticket" ],
  "order": "http://fx-orders.com/ordr321",
  "payment": "http://fx-orders.com/paym432",
  "params": [ "usd/jpy", "buylim", 81.7, 1000.00 ],
  "ask": 81.6,
  "status": "paid"
}

HTTP/1.1 204 No Content
```

Now the Order sees the update to the Ticket: all done, all paid. Again, if the Payment were POSTed directly to the Ticket, this Ticket state would probably form the POST response instead.

The Hypermedia Constraint

That example showed how Functional Observer REST works, and how it meets all of the REST constraints, apart from two constraints not yet covered: the Hypermedia Constraint and the related Stateless Constraint.

It turns out that the Hypermedia Constraint maps right on to the Functional Observer programming model.

Stateless Constraint

Stateless means that the server holds no ongoing state between client requests to track or in any way drive or co-ordinate a succession of such client requests or anything else about the client, beyond offering responses on demand, containing links to more resources. Each request is seen as independent by the server at the HTTP level, and any state held across requests is only in the state of its own resources.

In FOREST, object resources may establish a relationship with “client-side” objects, but the interactions, inter-links and state are out in the open in the content, not hidden away at the HTTP interaction level, or in any client-specific session state. The HTTP layer of FOREST only handles the mechanical mapping of Functional Observer pulls and pushes between domain-level object resources into one-shot, independent GETs and POSTs.

So if the server is not holding state for the client, the client has to do all of its own co-ordination in the face of the server resources it sees. A client has its own "application state" to guide its working through the hypermedia Web before it. Indeed, hypermedia is the "engine" of this application state in REST – the Hypermedia Constraint.

The Hypermedia Constraint thus works in hand with the Stateless Constraint, since server statelessness with respect to clients leaves its hypermedia as the only influence it has on client or application state.

Hypermedia Constraint

The phrasing of this constraint in Fielding's thesis is "hypermedia as the engine of application state", which he then goes on to elaborate: application state is the total state of the client, including current pages, open tabs, history, bookmarks, rendering of page, rendering of images, prefetching of links, etc. The state is stable once all requests are done with.

Hypermedia is the "engine" of this application state because the current state of a client application is directly dependent on the pages and images found by links, whether fetched automatically or by user action. Evolution of a client's application state is a non-deterministic, heuristic, parallel and user-driven exploration of the server-side hypermedia landscape. So you could say "hypermedia and the *user* are engines of application state". The hypermedia content and structure simply "is" – and it is entirely up to a client, including the user, what it sees and where it goes to see it and when and in what order. That hypermedia structure does not imperatively dictate the evolution of application state, it declaratively guides or constrains it.

This even applies to the Web crawler, which will automatically evolve its "crawler application state" as it jumps from link to link. This application state is an index of pages it discovers. Now, when re-published as a search engine's results pages, this application state becomes hypermedia itself. Hence, in this case, we have "hypermedia as the engine of *hypermedia*"!

The benefit of this constraint is that it encourages a declarative evolution of system state, rather than servers statefully walking clients through their imperative agendas. Servers send some hypermedia then forget the client even exists. Given a hypermedia type, the client can dynamically make its own choices without prior client–server coupling.

The Hypermedia Constraint is also related to Client–Server in the sense of separation of concerns and independent components. Finally, both the Hypermedia Constraint and Self-Descriptive Media Types deliver loose coupling: clients are free to evolve both their own application states at run-time and their interpretation of incoming data independently of servers, and servers can similarly evolve their hypermedia within its type, or evolve the type itself, in a backwards-compatible way, independently of clients.

Applying the Hypermedia Constraint to the Fulfilment Scenario

The Order server provides hypermedia to the Fulfilment server, in the form of an Order which links to a Payment, so in that case, it is the Fulfilment server whose application state we want to be dependent on this Order server hypermedia. The Fulfilment server is akin to the search engine crawler in this way.

So what exactly is the "application state" in the Fulfilment server?

Consider what depends on the Order server's hypermedia: the Fulfilment server's Ticket, corresponding to the Order it has been given, is dependent on both that Order and its linked Payment. So the Fulfilment server's Tickets *are also its application state*. There is no other state that is significant here. The Ticket can hold links to the other object resources that it is working on or cares about; Tickets hold links to Orders and Payments.

Driving Ticket lifecycle is the entire purpose of the Fulfilment server, just as rendering documents is the entire purpose of the browser, and collecting and indexing pages is the entire purpose of the search engine crawler.

And like the crawler through the search engine server, we can publish the application state we have evolved to – the Tickets – as more resources, more hypermedia.

For us in our fulfilment scenario, the Hypermedia Constraint quickly boils down to: the current state of a Ticket depends on the state of its Order and Payment. And, in the other direction, when the Order server is being the client, the Order depends on the Ticket, to see how it is being fulfilled and to make the payment when it is ready.

Which, of course, is another way of describing the Functional Observer Pattern.

Hyperdata as the Engine of Hyperdata

In other words, we meet this REST constraint in FOREST simply by following the Functional Observer model, ensuring that the states of the objects in our hyperdata graph published by our integrated applications are dependent on the states of other objects around them, discovered through links. HTTP is used as the pipework of that engine, to move object state around.

Or, to paraphrase for FOREST: "Hyperdata as the Engine of Hyperdata". Which is exactly the Functional Observer model. Our use of Functional Observer, when distributed, gives us the Hypermedia Constraint, which, when distributing over HTTP, pretty much delivers all of the remaining REST constraints, too.

We have an interacting, interdependent, interlinked object Web. Each object in that Web is independent while also *inter*dependent: the master of its own destiny, deciding how to evolve by itself, based upon the hyperdata context it finds itself within – the context set by other, similarly independent yet interdependent objects.

Of course, some state has to be dependent on incoming events or external processes, rather than being entirely driven from internal object interactions, as in our example, where Tickets depend on the Foreign Exchange market. However, all internal interactions should work through the interdependencies of our interlinked objects in order for FOREST to conform to REST.

The Role of POST

The Hypermedia Constraint is largely GET-oriented. The Fulfilment server can GET the Order that a Ticket it hosts links to, or its linked Payment, from the Order server, because that Ticket depends upon them. The Order server may GET the Ticket, because the Order depends upon it.

However, in Functional Observer, we have push by POST as well as pull by GET. As we have seen, the Order server will also POST the Order directly to the Ticket on the Fulfilment server; and, indeed, that is what one would expect to be the normal flow of events when one server has something to say to the other. It is more efficient to push than to pull or poll when things are changing.

But how does such a POST of an object resource representation fit into the REST constraints? What are the responsibilities of POST in this scenario and in this interacting object Web interpretation of REST?

As already noted, REST has little to say on the actual methods we use. The general consensus in the REST community seems to be that POST can be used in pretty much any way you like, if you are not using its data-editing interpretation as "create a new entry in a collection".

Since REST has little to say on the subject of the workings of POST, we could look at the HTTP spec, (Fielding et al. 1999), which currently says something about "subordinates", alluding to the create-new role, but also has a catch-all "data-handling process" function, which is not especially useful to us. This is one of the areas that may be clarified by the HTTPbis working groups (Fielding et al. 2010).

So, we turn finally to the way the Web works, and it all actually becomes very simple, even to fit POST back into the REST terminology.

POST in REST Terms

On the Web, what we can POST, and to where, is set up for us with forms. Forms supply domain-specific annotation and structure, as well as a URL to POST to. Presenting a form and having the user fill it in is entirely within the realm of application state. That we got a form at all is an example of hypermedia being an engine of that state.

This application state then makes its way back to the hypermedia realm through the submit URL, and in any ongoing redirects and pages that depend on the submitted form.

Thus, in REST terms, POST is used when certain application states are reached, to send a little of that application state back into the hypermedia graph. That hypermedia may then change, in resource state or link structure, to drive our application state in a different direction. It is "hypermedia as the engine of an element of application state that then acts as an engine of hypermedia"!

So, in our scenario, where application state comprises first-class object resources, this becomes even more straightforward. The Order server knows, in the face of Ticket hypermedia or hyperdata, that it can POST an Order, as its submission of a part of its own application state, to the Ticket on its URL. Then the Ticket and its local hyperdata can change as a result of this POST; perhaps then driving the Order application state that depends on it in a different direction. Similarly, the Ticket can be POSTed into Order hyperdata, which may also react.

Logic Drives Push Between Interdependent Objects

The Order server knows that it can push the Order to the Ticket because it is part of the domain interaction specification for Order and Ticket objects – the Order itself drives the push of its own state. This is the equivalent in automatic business or domain logic of a human driving an interaction, reading and filling in forms. The logic that determines the dependency of an Order on a Ticket is the same logic that determines that that Ticket is now likely to be interested in being POSTed the current Order state.

Indeed, that the Ticket may itself be dependent on that Order. From the Ticket's point of view, as application state that depends on Order server hyperdata, it is continuing to meet its obligations under the REST Hypermedia Constraint to be dependent on the Order hyperdata, but instead of using GET to pull it, it is being pushed the Order, including its own URL for future GETs and perhaps with a hyperlink to the Payment.

It runs its own domain logic over the hyperdata graph visible in front of it, and this logic is indifferent to whether that hyperdata was pulled or pushed into view. Of course, once an object with its URL has been seen from a push, it is always possible to pull or poll on that URL, as long as it is saved as a link back.

Thus the initial POST is the key event – once the target has received that, it takes over responsibility for its dependency on the incoming object. Subsequent POSTs are simply timely notifications of state change when the pushing object wants to take responsibility for that. Otherwise, the target can poll when it is interested.

Optimising POST

That concludes the description of the mapping between Functional Observer and REST's Hypermedia Constraint over HTTP, via the mapping from pull and push observation to GET and POST. It is reassuring that this often tricky constraint of REST can be met easily by using the simple and powerful Functional Observer object programming and interaction pattern.

It is worth now exploring the role of POST as idempotent object push in FOREST, as there are some optimisations that can be made, knowing how it is used.

POST as Cache Push

We have a clear role for POST in FOREST: to push an updated object state to other objects that depend on it – an object that is actually hosted by the source of the POST request and could otherwise be pulled or polled to meet those dependencies.

In this role POST is now idempotent, as it is effectively a "reverse GET". We can push an object's current state as much as we like in the same way that it could be repeatedly fetched idempotently by GET. To enable this idempotency and symmetry, the POSTed object must of course be self-descriptive enough to include its own URL.

Thus in FOREST a POST request is much the same as a GET response: it has a Content-Length, Content-Type and body. Since it carries the POSTed object's URL, it can be pushed into the client-side cache of the target's server for future GET requests. So to finish off this "POST request as GET response", it could carry cache information, too.

The main cache parameters needed are Etag and max-age. These could go alongside the URL in the object itself, in a metadata section:

```
POST /ordr321 HTTP/1.1
Host: fx-orders.com
Content-Type: application/json

{ "%url": "http://fx-broker.com/tick111",
  "%etag": 1, "%max-age": 10,
  "tags": [ "fx", "ticket" ],
  "order": "http://fx-orders.com/ordr321",
  "params": [ "usd/jpy", "buylim", 81.7, 500.00 ],
  "ask": 81.9,
  "status": "waiting"
}
```

But, as mentioned above, these metadata parameters are candidates for pushing down into HTTP headers; into the POST request headers, unconventionally, with the URL as a "Content-Location" header:

```
POST /ordr321 HTTP/1.1
Host: fx-orders.com
Content-Location: http://fx-broker.com/tick111
```

```
Etag: "1"
Cache-Control: max-age=10
Content-Type: application/json

{ "tags": [ "fx", "ticket" ],
  "order": "http://fx-orders.com/ordr321",
  "params": [ "usd/jpy", "buylim", 81.7, 500.00 ],
  "ask": 81.9,
  "status": "waiting"
}
```

Of course, this may cause issues, as unfortunately Cache-Control has a different meaning on a request: it could be seen as trying to constrain the POST response, rather than describe the POST request body or entity. If that is an issue for you, you could use the "Expires" header instead.

Note that, since such cache-pushing is an optimisation – it is always possible to simply GET the object once it is been seen – these cache parameters are optional, and could be ignored if discovered but not understood in the POST headers.

Optimising for Multiple Dependents on the Same Host

Now, what if there were two objects in one host that both depended on the same remote object? The framework driving those two objects could clearly save time and bandwidth by fetching the remote object dependency just once for both of them. It could put that object into the client-side cache, then show it to both dependents. It could then re-use the cached object immediately if a third object also looked at it.

Things get a little more complicated when the dependency pushes itself at those two objects. After doing this the first time, targeting each dependent, it would be wasteful to carry on POSTing twice per update into their common host.

Instead, it would be better if the object were pushed once, at one of the two objects, and, as in the GET case, put into the client-side cache, then distributed to the other dependents. That one POST request could be directed at the URL of one of the dependents at random, or at the dependent whose expected response is most interesting or needed most promptly.

But how does the host or framework of the remote object know that two of its dependents are on the same host, or share the same client-side cache? You cannot just assume two URLs with the same host:port actually point to the same back-end host; that could be a reverse proxy.

One solution is to return a unique cache identifier in the "Server" header of POST responses, to allow such correlation and bundling in future. Another solution is to add a new header in POST responses, perhaps "Cache-Notify", which not only serves the same purpose of uniquely identifying that shared cache, but does that by offering a common URL that can be POSTed to, for all objects that return it.

The object returned from a POST to this Cache-Notify URL would now be in the hands of that server – perhaps the first dependent notified that reacted or updated. Almost equivalent to Cache-Notify would be a URL set up per incoming object – effectively its URL in the cache. Then you *would* actually use PUT to set the state of that cache "resource", and would not need the URL on the POSTed object itself.

For un-observing, in the first approach, a 403 could be returned if the object chosen lost interest, meaning the next POST should go to the next dependent sharing that Server header. In the second and third approaches, a 403 would mean there were no more dependents of the POSTed object currently left on this host, being served by that Cache-Notify or per-object URL.

All of this is just optimisation that would be implemented by the framework to keep it transparent to the object programmer, and none of it is required in FOREST.

Finally, the "Cache-Notify" header can be added to GET requests, to request a "subscription" to future changes in that target URL. This is a host-level agreement that does not involve any individual objects; the subscribed object would be unaware of the propagation of its updates. It is thus quite a different thing from the object-initiated form above. Indeed, it amounts to the generic form of network Publish-Subscribe. Fielding has rejected Publish-Subscribe on the Web (Fielding 2008), and this is an optional optimisation in FOREST that can be used to keep cluster caches fresh, etc.

Asymmetric "API"s

Sometimes you really are in an asymmetric situation and your clients are not able to host their own objects. Then the server side of this becomes more like an "API". FOREST can support this asymmetric style of server-centric APIs.

The server carries on publishing its own Web of cacheable objects for the client to GET. The client can then continue to use Functional Observer to maintain the dependencies of its unpublished objects on those server-side objects.

But what about POSTs of those client objects, when there are no client URLs to include? How do server objects know that a POSTed object is the one they depend on and that they've seen before, without a URL to identify it?

The trick is simple: instead of the client sending the URL of its object in a POST, it just sends its unique id. Let us call that a "UID", and say that it could be a UUID or a GUID. Here is what the Order POST would look like now:

```
POST /dealer123 HTTP/1.1
Host: fx-broker.com

{ "%uid": "uid-a321-6fb3-129af3d",
  "tags": [ "fx", "order" ],
  "params": [ "usd/jpy", "buylim", 81.7, 500.00 ],
  "dealer": "http://fx-broker.com/dealer123"
}
```

```
HTTP/1.1 200 OK
Content-Location: http://fx-broker.com/tick111
Etag: "1"

{ "tags": [ "fx", "ticket" ],
  "order": "uid-a321-6fb3-129af3d",
  "params": [ "usd/jpy", "buylim", 81.7, 500.00 ],
  "ask": 81.9,
  "status": "waiting"
}
```

The UID is prefixed with "uid-" to make its "type" immediately clear. Now, as we see here, with this approach we have to return the Ticket for this Order immediately. The rest of the interaction is much the same, only with more polling:

```
POST /tick111 HTTP/1.1
Host: fx-broker.com

{ "%uid": "uid-a321-6fb3-129af3d",
  "tags": [ "fx", "order" ],
  "params": [ "usd/jpy", "buylim", 81.7, 1000.00 ],
  :
}

:

GET /tick111 HTTP/1.1
Host: fx-broker.com
If-None-Match: "2"

HTTP/1.1 304 Not Modified
Etag: "2"

:

GET /tick111 HTTP/1.1
Host: fx-broker.com
If-None-Match: "2"

HTTP/1.1 200 OK
Etag: "3"

{ "tags": [ "fx", "ticket" ],
  "order": "uid-a321-6fb3-129af3d",
  "params": [ "usd/jpy", "buylim", 81.7, 1000.00 ],
  "ask": 81.6,
   "status": "filled"
  }

  :
```

At payment time, you POST the Order, then the Payment:

```
POST /tick111 HTTP/1.1
Host: fx-broker.com

{ "%uid": "uid-a321-6fb3-129af3d",
  "tags": [ "fx", "order" ],
  :
  "payment": "uid-03de-008a-eff20d7"
}

HTTP/1.1 204 No Content

:

POST /tick111 HTTP/1.1
Host: fx-broker.com

{ "%uid": "uid-03de-008a-eff20d7",
  "tags": "payment",
  "order": "uid-a321-6fb3-129af3d",
  :
}

HTTP/1.1 200 OK
Content-Location: http://fx-broker.com/tick111
Etag: "7"

{ "tags": [ "fx", "ticket" ],
  "order": "uid-a321-6fb3-129af3d",
  "payment": "uid-03de-008a-eff20d7",
  :
  "status": "paid"
}
```

Hence, there is a "pseudo hyperdata" coming from the client, since it cannot publish any objects. The server may cache those pushed objects keyed by their UID, and present a programming model that makes them look the same as other, published object resources.

On the subject of asymmetry, note that it is also theoretically possible to tunnel a 100% RESTful state-transfer protocol backwards through HTTP using various approaches.

Data Editing API

You may think you need an API to allow direct editing of a server's resources. But in the majority of cases, it is better to think a level higher – to think in terms of what you actually want to do in domain or application terms, and design interacting objects that declare their own higher-level meaning and intents. That is where the Functional Observer and FOREST approaches work best.

But what if you *really do* want a data editing service, offered through a traditional, asymmetric API? Using HTTP RESTfully using the traditional, four-method protocol style typified by AtomPub runs the designer into issues of ownership and partial ownership of data, responsibility for integrity, server-driven hypermedia, etc. For example, when suggesting an edit of a server resource using PUT, the client has to round-trip the entire representation it received, with all the server's links, and all the content that it may not even understand. It drops its own edits somewhere in the middle and then lets the server sort it all out.

FOREST offers an alternative solution. But since FOREST pushes all the domain-level interactions up into the content in Functional Observer exchanges and only uses HTTP for pulling and pushing data, it is not obvious how to create, update and delete data, or how to manage optimistic locking. These are all traditionally performed in REST approaches through PUT, DELETE, PATCH, If-Match/Precondition Failed, etc., in the HTTP layer. What are the equivalents to these up in the content or domain layer, where FOREST has just state declarations – which are anyway incompatible with such imperative editing commands?

There is limited space in a single chapter to describe the full interaction here, but to summarise it: we should agree on a schema or syntax for describing "idempotent edit intentions" in objects that are then POSTed to a target object to be edited. These declarative intentions say things like: "I think you are in this state, and I'd like you to move into this state", or "this is what state I want you to have, regardless of your current state", or "please have this bit of data, that I believe is here currently, appear over there, instead". All of these, being idempotent, can be POSTed repeatedly; those that specify current state are simply ignored if there is no match.

The great advantage of handling data editing up in the domain or content layer is that that layer has full insight into the nature of the data being edited, and can thus make intelligent decisions about incoming requests. For example, it may be possible to merge a "late" edit request instead of rejecting it as out of sync, or perhaps to partially apply it – if within the integrity constraints and expectations of the domain. Otherwise, version synchronisation is still dealt with in the content in a similar way to the "not-as-ordered" flag in the fulfilment example above.

But then again, if you are thinking in such high-level domain terms, are you sure you want a low-level, direct, data-editing interaction, anyway? Just let the client express domain-level (rather than syntax-level) declarations and intentions directly!

User Objects

What if a user wanted to play in this object Web? Indeed, suppose that user, quite reasonably, wanted to use a browser to access it? How does FOREST's object resource interaction model work then?

In MVC terms, the raw domain Model objects of FOREST will need some decoration, some transformation into a View. Serialising them into XHTML would allow them to be rendered reasonably well in their raw state – then perhaps made more presentable by including some Javascript and CSS.

If serialised into JSON, they can be rendered to a View by a "FOREST browser" Web page, which would include some Javascript that knows how to navigate through and present the object Web, in from a starting point, perhaps detecting common object types and giving extra relevant interactivity for them.

On the POST side, or the Controller of MVC, the object in the application state that would be pushed back at the objects being rendered depends on those objects and their interaction specification.

The centre of any browser-hosted segment of object Web, however, is a first-class User object. This object can hold the user's identity, vCard, chat status, etc., any of which could be POSTed to objects that are interested.

This whole area is far too broad to discuss in any depth in this single chapter, as it takes us into the rich subjects of Web Applications, Widgets, Ajax, Forms, HTML5, Mashups, etc. Suffice to say that FOREST is an excellent foundation for all that.

Programming Functional Observer

Functional Observer object resources are masters of their own evolution, never told what to do by other objects in the hyperdata graph. They determine their own state by looking around. These objects take responsibility themselves to pull and push state, guided by the domain logic for the class to which they belong.

All the pulling and pushing can happen completely automatically and transparently – an object just has to focus on meeting its domain rules, and all observed objects visible through links, or links to links, are fetched when needed; plus, every time that object updates, its Etag is incremented and its state is pushed out to its local and some of its remote dependents.

This programming model brings the focus right down to what is important – an object taking care of meeting its own domain constraints and state evolution given what it can see of itself and others through its links to them, and letting the framework take care of the state transfers in and out needed to support that. It allows very expressive declarative and reactive programming styles to be used. As mentioned above, this has affinity to the models available in Clojure and Erlang.

A great advantage of leaving the state transfer up to the framework is that it can handle not only the cacheing of remote objects, but also making local objects look much the same as remote ones, apart from the handling of timeouts and retries. This is most effective with an asynchronous and reactive programming model. Treating local objects like remote ones eases the further partitioning of an application. Clearly, it would make sense, in this light, to de-serialise remote object representations into their "solid" equivalent objects in an object cache.

This declarative programming model also leads quite naturally to the "eventual consistency" optimisation seen in large-scale systems. Instead of a workflow being kept in lock-step through an imperative model, we can relax and let the system "settle when it is ready". A relative disposition of object states at any time is either fully resolved, or "in tension" – needing further state evolution to reach the domain constraints being applied to each object involved.

Design Guidelines for FOREST

FOREST can be characterised as an informal list of design guidelines:

Functional Observer:

- Implement domain logic as functional dependencies between objects' state.
- Ensure object structure conforms to common, shared standards.
- An object's next state depends on its current state plus the observed state of other linked objects.
- Ensure objects are masters of their own evolution guided by other linked objects.
- Discover these objects through links and links to links.
- Alternatively discover objects via initial incoming push, leading to a new link.
- Implement interactions using the Observer Pattern with observation via pull and push.
- Guide the pulling and pushing of interdependent objects by domain logic over object interaction specs.
- Push new state out to all observers and any objects that should be interested according to the domain logic.
- Ensure an object cannot see the list of its observers.
- Ensure objects only take what they need from their observed objects.
- Use timeouts on pull and push that depend on domain context.
- Consider using or writing a framework that automatically handles the pull and push of dependencies locally and remotely, automatically setting new Etags and propagating new state.
- Drive appropriate objects partly or fully by external processes and interfaces.

Hyperdata:

- Expose domain and application state via HTTP as your object resources with ids mapped to URLs.
- Choose a common, base, data-oriented object serialisation or representation format from XML, XHTML1/5 or JSON.
- Within that format, ensure your objects conform to standard or pre-existing schemas where possible.
- Render links between objects as URLs in the serialisation, using links of the Media Type, if any, or maybe using XLink.
- Thus link up objects within and across applications into a global object Web.
- Separate responsibilities in your applications by partitioning your object space.
- Treat local and cached remote objects the same, to ease ongoing partitioning.

State Transfer:

- Use HTTP statelessly and only as an object representation state transfer protocol.
- Use standard HTTP headers and return codes to drive state transfer only.
- Use GET to pull remote objects on which local objects depend, following links.
- Use POST to push updated object state idempotently to known dependent objects.
- When POSTing, include the object's URL in its serialisation or in the POST headers.
- Use polling, max-age, retries and timeouts on both GET and POST as required.
- Return any new state of the target on a POST response, whether 200 or 303.
- Otherwise, return 204, 403, 405, etc., as required.
- De-serialise remote objects and cache only their object forms.
- Drive optimal cacheing by using HTTP headers correctly; return 304s, etc.
- Use proxies and proxy-caches where beneficial.

Optional:

- Consider using declarative, functional, reactive or other asynchronous programming styles over the Functional Observer framework.
- Consider using eventual consistency where appropriate.
- Consider exploiting cache information on POSTed objects held either in the object itself or the POST headers.
- Consider, as an optimisation, consolidating cache notification using single POSTs to a Cache-Notify URL.
- Consider, as an optimisation, asking for cache updates on GETs using Cache-Notify.

In short: design your application as a graph of objects that interact through the Functional Observer Pattern: write your application domain logic such that the state of those objects depends on their current state plus the state of those other objects they can "see" through links. Transfer that state by either pull or push. Integrate or distribute applications by publishing objects into a shared object Web via ids mapped to URLs and state mapped to a generic Media Type, with inter-object links. Then map object observation by pull and push to GET and POST respectively.

Benefits of the FOREST Approach

Here is a list of advantages of choosing the FOREST approach:

- Functional Observer means focusing entirely on each object's point of view with respect to surrounding objects.
- Functional Observer enables easy declarative, functional, reactive, asynchronous programming styles.
- RESTfulness easily attained by distribution of the Functional Observer object interaction pattern using HTTP.
- Any "REST API" essentially drops out of the object interactions.
- No threading, concurrency and locking issues – objects manage their own state in a naturally parallel way.
- Easy to separate concerns or partition across servers – naturally parallel processing model without tight application boundaries.
- Symmetric distribution across hosts acknowledges clients as first class servers – and servers as clients.
- Similar code whether a local or remote interaction – just timeouts and retry logic may differ.
- Good separation of concerns between HTTP state transfer and domain-level Functional Observer interactions.
- Conflict resolution dealt with up in the domain or application by richer business logic, not at the HTTP level.
- HTTP given simple state transfer role; observation maps to just the widely-used GET and POST.
- Use of POST can be more efficient than polling; POST can update caches.
- Gives "mashability" by exposing objects and linking them up across servers or applications.
- Web-ready public state, in well-understood, data-oriented type such as JSON, XML or XHTML.
- Requires stable, well-understood object types, structures and schemas, encouraging interoperability, re-use and mashability.
- Simple linking model of unique object ids and inter-object links as URLs.
- Creates hyperdata or an object Web – efficient, semantic.
- Eliminates URL design as URLs are opaque object ids; object Web is traversed through content semantics.
- Object-based partitioning allows fine-grained cache control; only transfer data that is needed or that is changed.

- Allows easy implementation of eventual consistency models for partitioning and scaling.
- Allows independent evolution and loose coupling of each distributed application through the Stateless, Self-Descriptive Media Types and Hypermedia Constraints.
- Delivers scalability through the Cache, Layered and Stateless Constraints.

Conclusion

FOREST describes a model of object interaction called a "Functional Observer Pattern". FOREST's Web of objects interact by setting their next state as a domain logic Function of their current state plus the states of other objects on which they depend, near them in the Web, Observed by pull or by push.

FOREST is distributed by a simple "Symmetric REST" architectural style – using just GET and an idempotent POST to transfer object resource state in each direction between interlinked, interdependent objects from applications or servers being integrated or distributed. Clients host their own objects, or servers can become clients.

FOREST's Web of object resources can be described as "hyperdata". Its interpretation of REST's Hypermedia Constraint can be stated as the more symmetric "Hyperdata as the Engine of Hyperdata". This is the distributed form of the declarative object interdependence derived from the Functional Observer model.

FOREST can be seen as a symmetric interpretation of REST and Web Architecture for two-way, dynamic data scenarios, rather than the usually one-way, static documents of the Web. In general, when REST is re-interpreted for such SOA-like applications, the result can be called a "Resource-Oriented Architecture" or an "ROA". FOREST can therefore be seen as a way of building a simple but powerful ROA for application integration or distribution.

Through being based on the Functional Observer Pattern, both in-process and across hosts, FOREST adds a number of advantages beyond the usual benefits it derives from naturally enabling a fully RESTful distributed architecture – such as interoperability, scalability and evolvability. For example, FOREST is easy to program due to its declarative, functional, reactive and asynchronous nature. This enables business or domain logic to be encoded in terms of state dependencies and evolution. Further, it is easy to distribute, not just across application partitions, but across multicore, without the usual concerns around threads and locks.

FOREST can be implemented using traditional languages such as Java, but fits most easily and naturally over the programming models available in a language such as Clojure or Erlang, or over an asynchronous HTTP layer such as provided by Node.js.

References

Armstrong J., Williams M., Wikstrom C. and Virding R. *Concurrent Programming in Erlang*, Prentice-Hall, Englewood Cliffs, NJ, USA. (1996). Online: http://www.erlang.org

Berners-Lee, T., Fielding, R.T. et al. *httpRange-14: What is the Range of the HTTP Dereference Function?* (2010). Online: http://www.w3.org/2001/tag/issues.html#httpRange-14

Berners-Lee, T., Fielding, R.T. et al. *Uniform Resource Identifier (URI): Generic Syntax*, (2005). Online: http://www.ietf.org/rfc/rfc3986.txt

Cragg, D. *Web Objects Ask, they Never Tell | The REST Dialogues,* (2009). Online: http://duncan-cragg.org/blog/post/web-objects-ask-they-never-tell-rest-dialogues/

Dahl, R. Node.js, (2009). Online: http://nodejs.org/jsconf.pdf

Fielding, R.T. *Architectural Styles and the Design of Network-based Software Architectures.* Doctoral dissertation, University of California, Irvine, (2000). Online: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

Fielding, R.T. *Post to the REST-Discuss Mailing List,* (2006). Online: http://tech.groups.yahoo.com/group/rest-discuss/message/6613

Fielding, R.T. *Economies of Scale*, (2008). Online: http://roy.gbiv.com/untangled/2008/economies-of-scale

Fielding, R.T. et al. *HTTP/1.1, Part 3: Message Payload and Content Negotiation*, (2011). Online: http://tools.ietf.org/html/draft-ietf-httpbis-p3-payload-14

Fielding, R.T. et al. *Hypertext Transfer Protocol – HTTP/1.1*, (1999). Online: http://www.ietf.org/rfc/rfc2616.txt

Fielding, R.T. et al. *HTTP/1.1, Part 2: Message Semantics*, (2010). Online: http://tools.ietf.org/wg/httpbis/draft-ietf-httpbis-p2-semantics/

Hickey, R. *Clojure's Approach to Identity and State,* (2009). Online: http://qconlondon.com/dl/qcon-london-2009/slides/RichHickey_PersistentDataStructuresAndManagedReferences.pdf

Mealling, M., Denenberg, R. *Report from the Joint W3C/IETF URI Planning Interest Group: Uniform Resource Identifiers (URIs), URLs, and Uniform Resource Names (URNs): Clarifications and Recommendations*, (2002). Online: http://www.ietf.org/rfc/rfc3305.txt

Part III
Development Frameworks

Chapter 8
Hypermedia-Driven Framework for Scalable and Adaptive Application Sharing

Vlad Stirbu and Juha Savolainen

Abstract This chapter describes our experiences designing a solution for scalable and adaptive sharing of desktop and mobile applications, using a lightweight network-based system compliant with the REST architectural style. The system delivers consistency of the rendered user interfaces with the state of the application logic using a stateless networking substrate. We describe the architecture focusing on how to model the user interfaces as a set of web resources. Then, we present the prototype that implements the functionality as an extension of the Qt framework, which works with different Qt-based user interface toolkits. Finally, we present a multi-display and multi-user Texas Hold'em application that shows how the system is used in practice.

Introduction

Sharing the user interface of an application allow users to control applications from remote computers. Modern mobile devices with sophisticated capabilities challenged the traditional role of desktop or laptop computers as users' preferred devices. They now expect to access applications anytime and anywhere while the usage experience is optimized for the particular device.

Traditional approaches on sharing the user interface relied on transferring the content of the framebuffer or the drawing commands from the device running the application to the device rendering the user interface. These techniques do not provide appropriate results for mobile devices or consumer electronics, which typically have smaller displays and/or different interaction metaphors. Therefore, to improve the user experience, the shared user interfaces have to be adapted to the rendering device look and feel.

V. Stirbu (✉)
Nokia Research Center, Visiokatu 1, Tampere 33720, Finland
e-mail: vlad.stirbu@nokia.com

Motivation

Traditionally, application sharing relied on remote user interface protocols that export, the content of the framebuffer (e.g. like VNC), or the drawing commands for the Graphic Device Interface (GDI) (e.g. like X Windows system). With these approaches the user interface is rendered on the remote device as instructed by the server with little or no possibility of customization.

Application sharing emerged when direct access to computing devices offered by mainframes and servers was not easily available. However, modern applications are developed using sophisticated frameworks that typically use the Model View Controller (MVC) (Krasner and Pope 1988) design pattern and its derivatives that make the applications easily maintainable by separating the application logic from the user interface. However, remote user interface protocols are not aware of the sophisticated capabilities of development frameworks that applications are using. As a consequence, the applications do not known that the user interface is rendered on a remote device.

Our framework changes this assumption allowing the developers of desktop and mobile applications to explicitly customize the appearance and behavior of the user interface rendered on remote devices, see Fig. 8.1. The application and the client become a network based system in which we leverage the MVC features of the application frameworks and the REST architectural style to have scalable and adaptive application sharing. With this approach we go beyond the classical

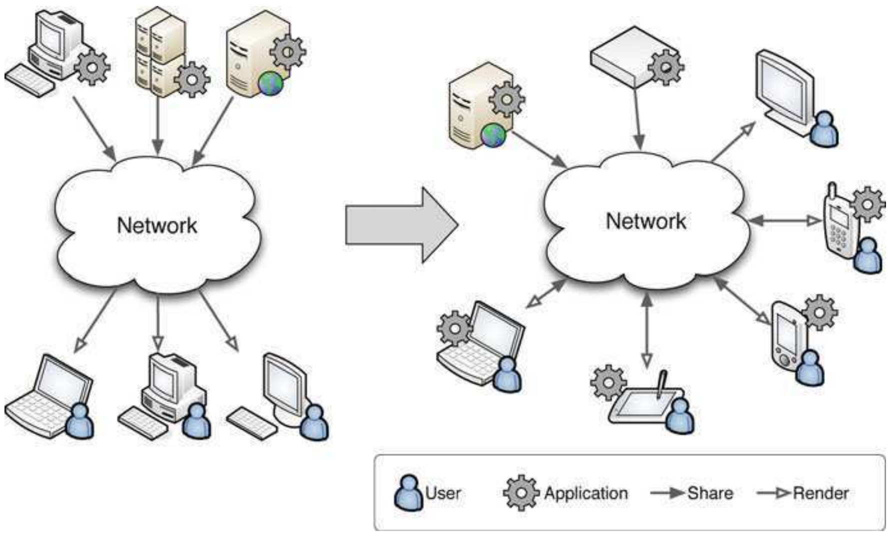


Fig. 8.1 Transition from traditional thin computing to an environment where applications can reside on any device and have the user interfaces rendered remotely according to the local look and feel

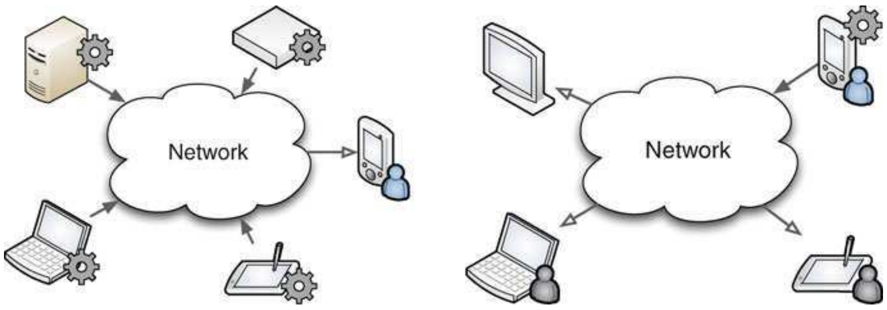


Fig. 8.2 Usage scenarios: application virtualization (*left*), and multi-display, multi-user applications (*right*)

application screen sharing scenarios in which the use interface is treated as a whole, enabling innovative applications that have the multiple user interfaces rendered remotely.

Usage Scenarios

The remote user interface paradigm can be applied in several ways depending on where the user and the application logic are physically located. From this perspective we can split the usage scenarios into two categories: *pull* and *push*. In the pull scenario, the user operates the device rendering the user interface, while in the push scenario the user operates the device where the application logic runs. These basic interaction primitives can be combined to tailor specific usage needs:

- **Application virtualization**. The application virtualization describes the situation in which the applications run on remote devices (e.g. personal computers, remote servers, consumer electronics or mobile devices) and the user interface is rendered on the device operated by the user. This scenario resembles closely the classical thin-client except that the exported user interface uses the local look and feel of the rendering device.
- **Multi-display applications**. The multi-display application scenario describes the situation in which the application runs on the device operated by the user and additional user interfaces are exported on nearby devices that render them using the local look and feel. Depending on the application context, these displays can act as secondary displays for the application or can be operated by additional users, a situation in which the application running on the remote device appears to the other users like a distributed cooperative application (Fig. 8.2).

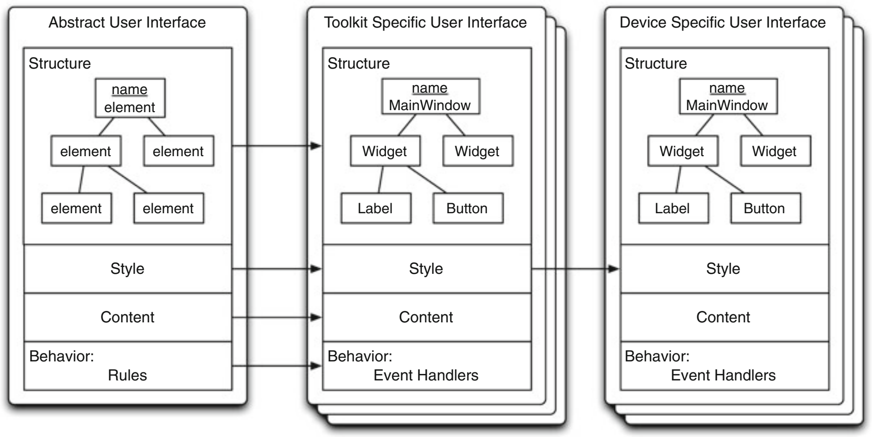


Fig. 8.3 Development process for scalable and adaptive user interfaces

Scalable and Adaptive User Interfaces

A scalable and adaptive interactive application is capable of exposing the user interface through multiple modalities and user interface toolkits, being able to adapt the user interface to the physical characteristics of the rendering devices. Developing adaptive and scalable applications is not trivial (see Fig. 8.3). To provide a good user experience the user interface needs to be designed with the particular device or device category in mind. Therefore, the user interface takes into account the features provided by the native user interface toolkit. A typical user interface can be divided into the following functional components: structure, style, content, and behavior. The structure contains the scene graph containing the elements of the user interface, with unique identification for each element. The style describes how the structure is presented to the users. The content describes which data is presented to the users in which elements of the user interface. The behavior describes what happens when the user interacts with specific elements of the user interface.

The scalable and adaptive user interface design process starts by creating the abstract user interface, which describes the user interface independently of any interaction modality or implementation technology. Its role is to capture the information that needs to be presented to the user, its structure and define the interaction behavior. Later, the abstract user interface is refined into toolkit specific user interfaces. They describe the user interface after a interaction modality has been selected (e.g. graphical). These user interfaces contain the final look and feel of the user interface by having the elements of the structure and the style mapped to the toolkit specific widgets, the content mapped to properties of the widgets, the layout of the widgets and behavior rules converted into event handler stubs. The user interfaces can be further refined for particular devices by adjusting the style parameters.

User interface description languages, such as UIML (Helms et al. 2008) and UsiXML (Limbourg et al. 2005), attempt to describe the user interface in declarative terms, without using low-level computer code. They aim at reducing the development effort by providing abstractions and automating the design process. Alternatively, SUPPLE (Gajos and Weld 2004) proposes a mechanism that automatically generates user interfaces according to device constraints. However, in practice, the challenges of having applications able to export the user interfaces to any device are significant (Want and Pering 2005). Therefore, we adopt a more relaxed approach that allows application developers to decide on what methodology to use for creating the user interfaces, and which devices they target.

From the User Interface to Web Resources

This section describes our approach for modeling the user interfaces of desktop and mobile applications as web resources. We start with an overview of the distributed system that enables applications to export their user interfaces to remote devices using the web architecture. Then, we describe in detail the functionality of the web resources that expose the user interface. Finally, we present the web-based mechanism that allows the remotely rendered user interface to be consistent with the state of the application, considering both the static dimension (e.g. user interface structure, consisting of elements and layouts), and the dynamic dimension (e.g. the values presented to the users at run-time) of the user interface.

Architecture Overview

The Model-View-Controller architectural pattern (MVC) separates the application engine that handles the data (e.g. the model) and the logic from the user interface that presents the model data, in a form suitable for the end user (e.g. the view), and handles the user input (e.g. the controller). The MVC pattern enables an interactive application to have multiple simultaneous views of the same model, allowing us to adapt the user interface to the various characteristics and form factors of the rendering devices.

The application sharing experience is provided by a system of two cooperating applications. The application to be shared provides the functionality of the model and the controller, while the user agent, which renders the user interface, provides the functionality of the view. In this distributed environment, the original interactive application becomes a network based system, which extends the classical MVC pattern with an event based mechanism that provides a level of consistency close to the case when the model, the view and the controller reside on the same physical device.

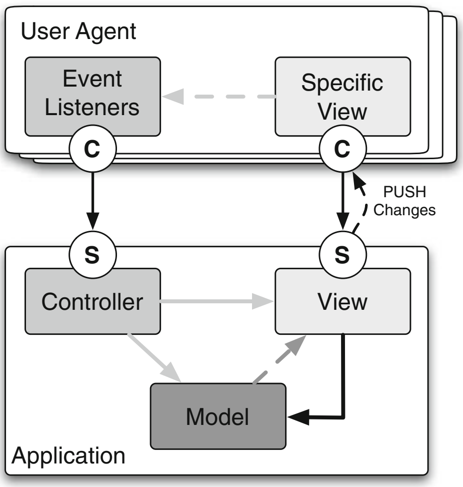


Fig. 8.4 Conceptual view of the Remote Model-View-Controller architecture

The Remote-MVC (depicted in Fig. 8.4) (Stirbu 2010) is based on a resource oriented architecture in which the user interface is exposed as a set of resources (e.g. view and controller resources). Each resource has a unique URI, is accessed using the methods of HTTP protocol and provides representations in well known formats such as XML specific to the rendering device or JavaScript Object Notation (JSON) (Crockford 2006).

The View-Related Resources

The User Interface

The user interface resource provides the first interaction point between the user agent and the application exporting the user interface (Fig. 8.5).

The `GET` method allows a user agent to acquire a user interface toolkit or device specific representation of the user interface. The user agent informs the user interface resource about its capabilities in the request using the `User-Agent` header and the HTTP content negotiation mechanism (e.g. the `Accept` header). The `User-Agent` identification string provides hints on the user interface toolkit version and on what device the user agent runs on, which can be further mapped to device form factor. The `Accept` header indicates the format in which the user agent accepts the representations. Based on the information provided by the user agent the resource implementation selects the most appropriate user interface, if any. The response body contains the toolkit or device specific representation of the user interface. Additionally, the header section contains links to the user interface element resources, to the event listener resources and to the monitor resource, which

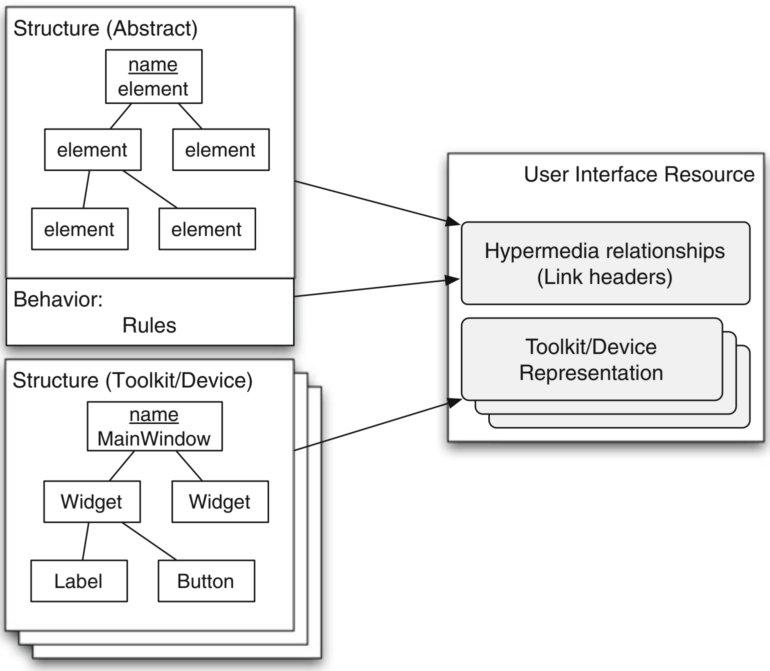


Fig. 8.5 Relations between the abstract user interface, toolkit/device specific user interfaces and the user interface resource

enables the user agent to be notified when the resource state changes. The URIs indicating the resources are provided using the `Link` (Nottingham 2010) header, and its relation `rel` to the current document are `widex.ui`, `widex.el`, and `monitor` (Roach 2010), respectively:

```
# Request
GET /appui HTTP/1.1
User-Agent: {User agent identification string}
Host: example.org
Accept: application/vnd.com.example.toolkit

# Response
HTTP/1.1 200 OK
Content-Type: application/vnd.com.example.toolkit
Link: </appui/uiHub/{uiElement}>; rel="widex.ui";
   uiElements="aUiElement,anotherUiElement,",
   </appui/elHub/{eventListener}>; rel="widex.el";
   eventListeners="anEventListener,anotherEventListener,",
   </monitor>; rel="monitor"

<!-- toolkit/device specific user interface representation -->
...
```

The User Interface Elements

A typical user interface is represented using a scene-graph data structure. Each node in this data structure represents an element of the user interface (e.g. a widget in graphical user interfaces), and each edge represents a parent–child relationship. Often, a node may have several children but only one parent.

The user interface of an application is determined by the structure of a scene-graph and by the properties of each node. We expose the application internal data structure that contains the view using a set of resources. By convention, we identify each resource corresponding to a user interface element using the following URI template:

```
http://example.org/appui/uiHub/{uiElement}
```

Although the URI scheme that identifies the user interface resources is flat, a typical toolkit specific representation of the user interface representation contains all information that allows a client to reconstruct the scene-graph structure. In case a toolkit specific representation does not have native support for describing the hierarchy of the user interface, we can overcome this limitation using use the XML linking language (Xlink) (DeRose et al. 2001) defined mechanism to encode the relationships between the resources.

The `GET` method allows a user agent to acquire the runtime values of relevant properties of the target user interface element. The properties of interest are indicated as the value of the `q` parameter of the query. If the query is missing, the server returns a list with all properties and their values that are considered relevant for the target user interface element:

```
# Request
GET /appui/uiHub/{uiElement}?q={paramName} HTTP/1.1
Host: example.org

# Response
HTTP/1.1 200 OK
Content-Type: application/json
Link: </monitor>; rel="monitor"
Link: </appui/uiHub/{uiElement}>; rel="edit"

{"paramName": value, }
```

If the value of a property has binary representation (e.g. an image), the response does not contain the value, but includes a JSON encoded link pointing to the content:

```
{
   "paramName":{
      "link":{
         "href":"/static/14411520525572 5056.png",
         "rel":"widex.static"
}}}
```

The `POST` method allows user agents to update the values of specific properties of the target user interface element. Typically, the argument of the call is a dictionary containing key-value pairs:

```
# Request
POST /appui/uiHub/{uiElement} HTTP/1.1
Host: example.org

{"paramName": value, }
```

The Controller-Related Resources

During typical usage, a user of an interactive application generates a large number of events. Among them only a small number are relevant for the application. Although all events emitted by the local window manager are passed to the application, the event handlers treat only the relevant ones, the rest being either ignored or handled by the widget implementations. For example, an application might be interested only when a button is pressed. This application has only one event handler (e.g. on_button_pressed) that handles the pressed event emitted by the button. The button widget implementation provided by the UI toolkit handles transparently the mouse movement through mouse move events and emits the pressed event only when the mouse is over the button and the mouse left button is pressed.

In our environment, it is not practical to transfer all events from the device rendering the user interface to the application host device, because the application handles there only a few of them. Each event handler defined in the controller is exposed as a correspondent event listener resource. The relevant listeners are provided to the user agent in the response to the user interface resource request. They are identified using the following URI template:

```
http://example.org/appui/elHub/{eventListener}
```

The `PUT` method allows a user agent to inform the controller that an event relevant for the target event listener was emitted on the user agent. The Controller is notified immediately when the message is received and the appropriate event handler is invoked with the provided parameters. Typically, the message body contains a list containing the values captured on the user agent by the listener:

```
# Request
PUT /appui/elHub/{eventListener} HTTP/1.1
Host: example.org

[aValue, anotherValue, ]
```

The Change Propagation Mechanism

Maintaining the state of the user interface synchronized with the application logic state is essential for an interactive application. In our network-based system, we use a change propagation mechanism to keep the user interface rendered by the user agent consistent with the application. The mechanism is driven on the server side by a special resource that notifies the user agent when representations change, and on the user agent side by the links embedded in responses from view and controller related resources. We first describe the monitor resource and then we describe how the change propagation effect is achieved in a way compliant with the REST architectural style.

The Monitor Resource

The view of an interactive application updates to reflect changes in the underlying model data. These changes are difficult to propagate to the user agents using only the request–response interaction pattern of the HTTP protocol, unless we use polling or long-polling techniques. However, these are expensive for the server who has to maintain open network connections for each user interface element resource. Instead we use a special resource that is able to stream notifications to the user agents whenever the representation of a user interface element changes, informing also how to obtain the change.

The `GET` allows the user agent to receive notifications whenever the user interface element resources change. The server streams these notifications over a long lived connection, borrowing characteristics from the `WATCH` method introduced in ARRESTED (Khare and Taylor 2004). Each notification is a link encoded in JSON representation (Allamaraju 2010). The user agent receiving such a notification performs a `GET` request on the target URI provided:

```
# Request
GET /appui/monitor HTTP/1.1
Host: example.org

# Response
HTTP/1.1 200 OK
Content-Type: application/json
Transfer-Encoding: chunked

{
   "link": {
      "href": "/appui/uiHub/aUiElement?q=aParamName",
      "rel": "widex.update"
}}
...
```

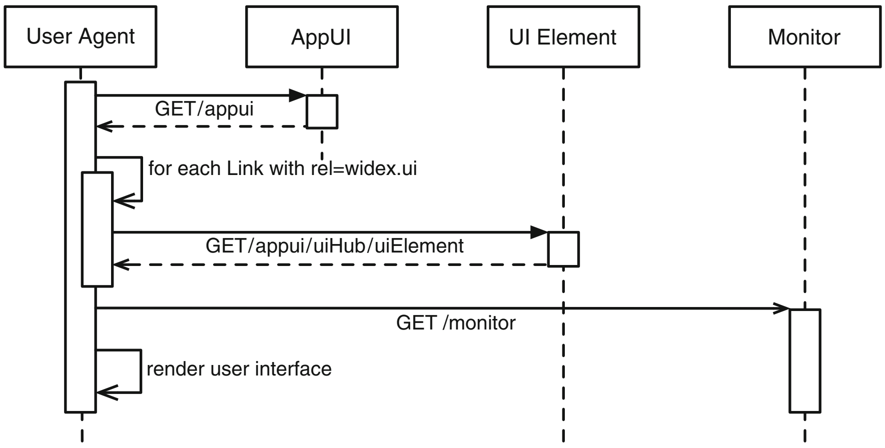


Fig. 8.6 Interaction pattern during the initialization of the change propagation mechanism

Orchestrating the Change-Propagation Mechanism

The change propagation effect is obtained as a result of cooperation between the user agent and the interactive application. The interactive application notifies the user agent when the content presented by the user interface elements change due to updates in the model, and the user agent notifies the application when the user edits the content presented by the user interface elements (e.g. edits content of fields in a form) or when the user interacts with specific user interface elements (e.g. clicks a button). The first phase of this process consists in the initialization of the user interface on the user agent side. The second phase involves the propagation of changes as they occur on the application side or on the user agent side.

The interaction pattern of the initialization is depicted in Fig. 8.6. At this stage, the user agent acquires a toolkit specific representation of the user interface, together with information that allows the user agent to acquire the current content presented by each user interface element, and which event listeners are relevant. The initialization ends when the user agent acquired the needed information and the user interface in its current form is displayed to the end user. The initialization process resembles a publish/subscribe scheme in which the application publishes the relevant resources and the user agent subscribes only to them.

The interaction pattern of the application triggered change propagation is depicted in Fig. 8.7. Whenever the content presented by a user interface element is updated by the model, the monitor resource notifies the user agent. The user agent acquires the new representation of the content by following the link provided in the notification and updates the rendered user interface with the newly acquired content.

The interaction pattern of the user agent triggered change propagation is depicted in Fig. 8.8. Whenever the user edits the content presented by a user interface element, the user agent propagates the change to the application by updating the

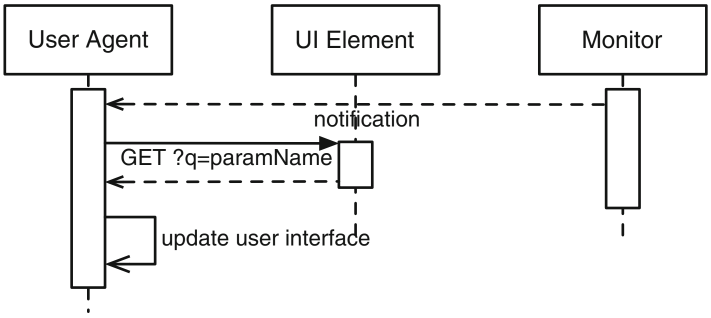


Fig. 8.7 Interaction pattern for application initiated change propagation

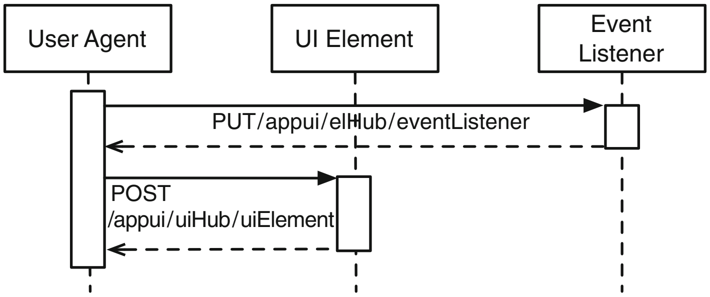


Fig. 8.8 Interaction pattern for user agent initiated change propagation

value on the corresponding resource exposed by the application. Similarly, the user agent notifies the application when a relevant event occurred by transferring to the application the event context characterized by the values corresponding to the event listener signature.

Prototype Implementation

This section describes the prototype implementation that enables scalable and adaptive sharing of Qt applications. Our implementation features the core components providing the server and the user agent functionality, and a set of add-on tools that speeds the task of creating and inspecting the network behavior of applications. The programming environment for application developers is Python, the bindings for Qt framework being provided by either PySide[1] or PyQt4.[2]

[1]http://www.pyside.org/.

[2]http://www.riverbankcomputing.co.uk/software/pyqt/intro.

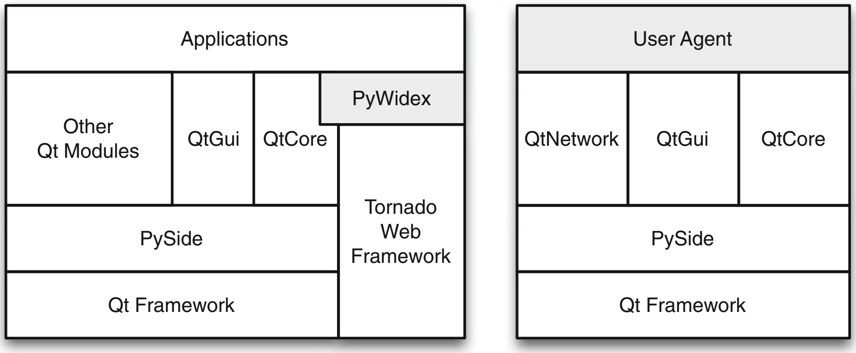


Fig. 8.9 The core components and their relation to the Qt software stack: application (*left*), and user agent (*right*)

The Core Components

The core components contain the basic functionality that enables scalable and adaptive sharing of Qt applications. The server functionality is provided by PyWidex (Widget Description Exchange). The package offers WebBackend, a convenience class that encapsulates all features needed to expose a view as a set of web resources, runs in its own thread, and is implemented as a web application running on top of Tornado,[3] a non-blocking, event-driven web server and RESTful framework optimized for real-time web services. Typically, an instance of this class is a property of the top level widget that provides the sharable view (Fig. 8.9).

The user agent is a standalone Qt application that is able to render the user interfaces exported by remote Qt applications, using the user interface toolkit specific to the rendering device. The user agent implementation is currently able to render user interfaces using multiple Qt user interface toolkits, e.g. QtGui or MeeGo Touch Framework. To render the user interface, the user agent application needs to know only the URI of the user interface resource of the view exported by the application. Once this information is known, the user agent configures itself using the information provided in the user interface representation.

Tools

Besides the core components, our prototype provides a set of add-ons that speeds the process of developing compatible applications:

[3]http://www.tornadoweb.org/.

- **Discovery**. The discovery functionality facilitates the creation and deployment of multi-display applications by enabling developers to specify on which kind of devices the user interfaces are to be rendered, and by allowing users to find instances of those devices in the proximity. A daemon running on the device allows remote applications to control the user agent. Currently the tool is based on zeroconf and support two user agent categories: public and handheld displays. Public displays automatically accept requests to render remote user interfaces while handheld displays are user devices, such as smartphones and tablets, which need user acceptance before a request to render the user interface is accepted.
- **Profiler**. The profiler tool provides tools for visualizing the interactions between the user agents and the applications.

Developing Multi-display and Multi-user Applications

In this section, we present the results of our experiments developing multi-display and multi-user applications using our middleware. We first describe how the test application works and then provide performance measurements.

Case-Study: Texas Hold'em Application

To demonstrate our middleware we implemented a Texas Hold'em application and used it in an environment containing consumer electronic devices (see Fig. 8.10) (Stirbu and Leppanen 2011). We have two users (e.g. Bob and Alice), each using a Nokia N900 smartphone, and a network enabled TV set having the back-end provided by a laptop running Ubuntu. In this setup, the TV set acts like a public display and Alice's N900 as a handheld display. All devices are connected to a local area network over Wi-Fi.

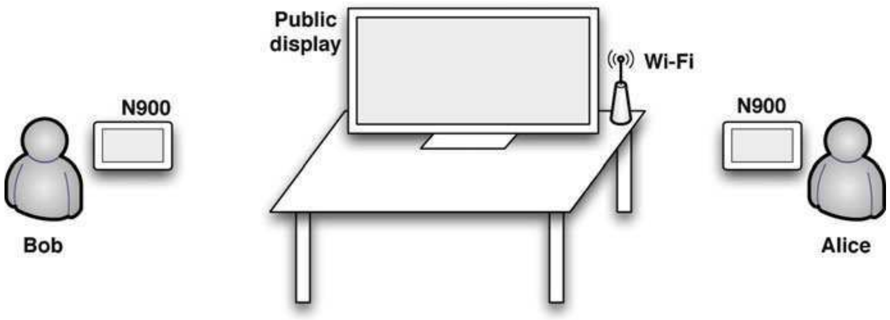


Fig. 8.10 Texas Hold'em: deployment environment

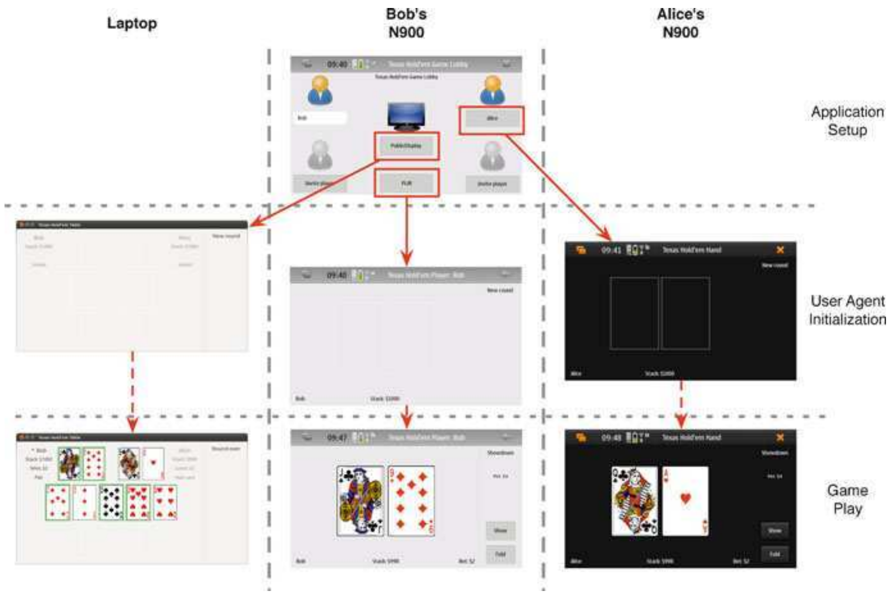


Fig. 8.11 Texas Hold'em: interaction flow

The Texas Hold'em is a multi-display and multi-user application. It has capabilities to accommodate up to four players, each presented with a view of their cards together with player specific information. Besides the player views, a public display presents the community cards and information of common interest about the game.

Figure 8.11 describes the typical usage scenario. Bob, the game host, starts the application on his handheld device. The application starts with the lobby view, which allows Bob to control where the views are displayed. First, he selects the public display, then invites Alice to join the game. The user agents on the public display and on Alice's handheld device connect to the corresponding table view and player view exposed on Bob's device. As the initialization is complete, the game can start. During the game, betting is controlled from the application engine by enabling and disabling the buttons in the appropriate player views.

Experimental Evaluation

The distributed interactive applications enabled by our system are presented to the user of each rendering device using the local look and feel. As each remote application looks like a native application, the users have similar expectations for the remote applications as for the local applications. In practical terms, the user interfaces of remote applications should have a reasonable startup time, and stay responsive in the intended usage environment.

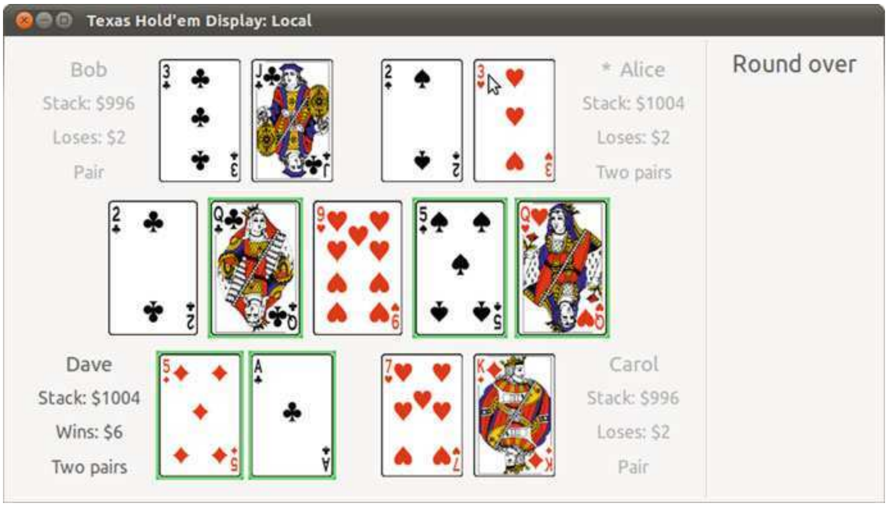


Fig. 8.12 User interface elements in the game table view

For any application, each user interface level interaction is typically implemented by the middleware using several network level interactions. For example, for initializing the user interface there is at least a request for each user interface element, which might add up to large numbers depending on how complex the user interface is. While using the application, each user interface change is implemented using one or two requests. Therefore, to evaluate the responsiveness of the user level interaction we need to aggregate the information from individual network interactions.

Among the views of the Texas Hold'em application, the game table is the most complex. The view presents information of common interest about the state of the game: four widgets for each user that plays the game, containing player specific information, one widget for showing the community cards and a side panel that provides information about the game phase. In total, the view contains 13 label widgets displaying images of the cards and 21 label widgets displaying text (Fig. 8.12).

The number of widgets that compose the game table makes this view a good candidate for determining how much time is needed by an user agent to render the initial user interface. Also, as the view contains common information about the state of the game, it gets updated frequently. Therefore, we use it for assessing how much time is needed to deliver the updates to the user agent.

We performed two round of tests. For the first test we used mobile devices for both Bob and Alice in an environment that simulates the intended usage scenario. For the second test we used laptops for all users with the intent of evaluating the impact of the hardware configurations.

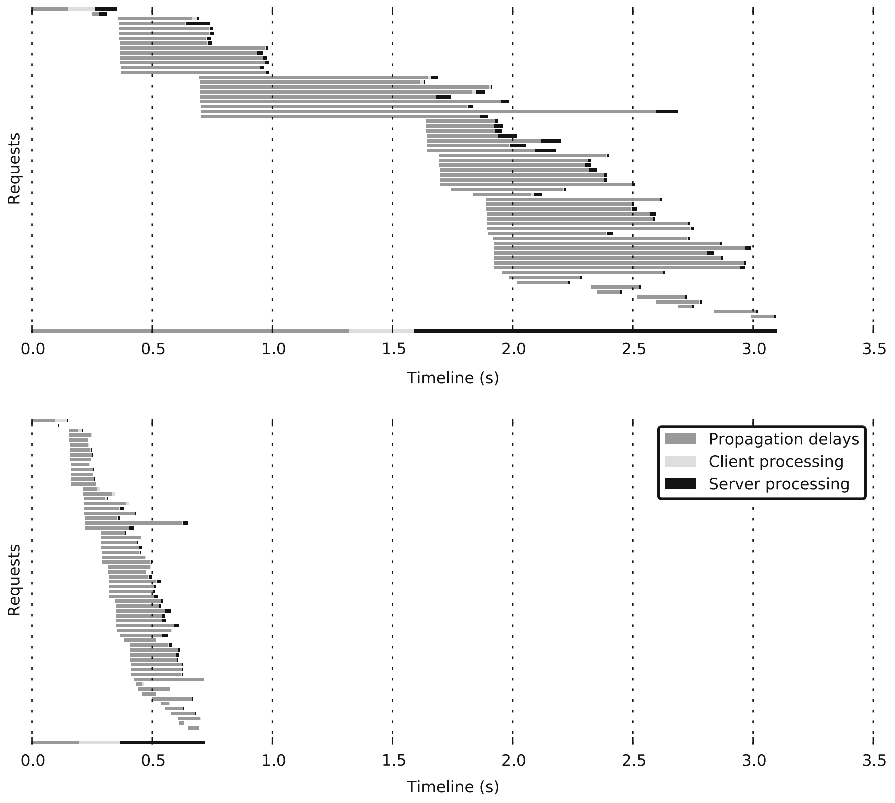


Fig. 8.13 Game table initialization: application running on N900 (*top*) and application running on laptop (*bottom*)

User Agent Initialization

While the user perceives the initialization of the user agent as one button click away, at network level the operation is decomposed into 67 requests. The interactions between the user agent and the application for initializing the game table user interface are described in Fig. 8.13. The top chart presents the interaction when the application runs on N900 while the bottom chart shows the interaction when the application runs on the laptop. Each interaction is broken down in three components each representing the time needed to process the request in the client, the server, as well as the delays induced by network propagation and the lower networking software stack.

The last bar in each chart represents the perceived initialization interaction. As both the server and the user agent are controlled by an event loop, operations associated with the requests are executed sequentially. This allows us to compute the total processing time for the client and server as the sum of individual request times. Due to HTTP pipelining, we are not able to determine the network induced

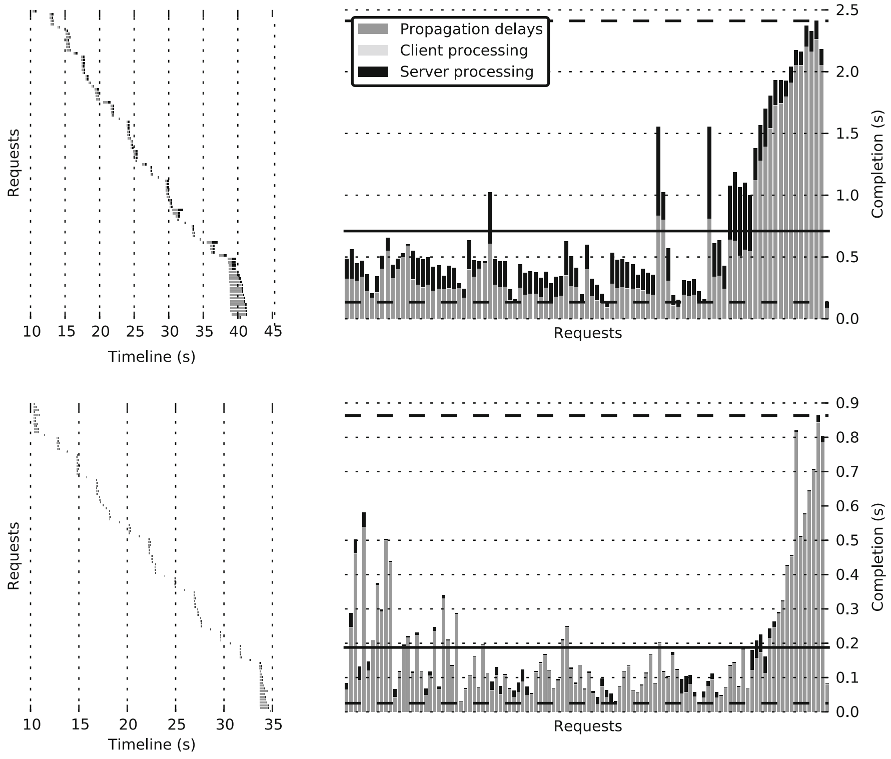


Fig. 8.14 Game table updates for a game round: application running on N900 (*top*), and application running on laptop (*bottom*)

delay, for completing the operation, directly from the information associated with individual requests. Therefore, we compute the perceived network latency as the difference between the time needed to complete the operation and the time used by server and user agent to process the requests.

User Interface Updates

We evaluated how fast the user interface updates are rendered on the user agent by having the game played by näive bots that advance to the next phase of the game after 2 s regardless of the cards in their hands. As for the initialization case, we first run the application on the N900 and then on the laptop. The user agents are the same in both test. The interaction pattern for each update is initiated by the application that notifies the user agent when a user interface element has changed, using the monitor resource. Then, the user agent request the new representation of the resource.

Figure 8.14 describes the interactions between the user agent and the application for both test scenarios as timeline and individual request completion time, each request being breakdown in time required by client and server to process the update, and network propagation delay. The beginning of the interaction represents the betting phase and is characterized by few updates, mostly related to one player at a time. The final part of the interaction corresponds to the showdown, a phase in which almost all user interface elements are updated in a very short time interval. This event resembles the user interface initialization, with the exception that the user interface structure is known and only the runtime values of the properties of the user interface elements are requested by the user agent, and the network propagation delay is longer due to notification delivery.

Discussion

REST is an architectural style. It contains a set of design guidelines, but it does not impose a specific architecture or a methodology. We follow these, but how RESTful is the resulting system? The Resource Oriented Architecture (ROA) (Richardson and Ruby 2007) introduces a practical approach for describing RESTful architectures through four concepts (e.g. resources, names as URIs, representations, and links between them), and four properties (e.g. addressability, statelessness, connectedness, and uniform interface). As the concepts are already covered, we'll focus the discussion on the required properties:

- *Addressability.* Our system exposes a resource for every piece of information that it serves about a sharable user interface: the structure using the user interface resource, the data presented to the user as user interface elements resources, and the behavior as event listener resources.
- *Statelessness.* Our goal of having the state of the rendered user interface synchronized with the application logic seems to go against statelessness. However, because the data of a sharable user interface is exposed at network level in a certain way by the system resources, the application web backend handles each HTTP request in isolation. All responses are generated based only on the information contained in the request. Having the networking substrate stateless, enables simpler generic implementation of the server functionality that can serve representations to user agents according to their needs.
- *Connectedness.* Resource representations in our system contain links to other resources. For example, the representation of the user interface resource has links to user interface elements and event listener resources, while user interface element resource may have links to static representations of binary content. These links point to subresources in the same realm, therefore the system is internally connected. Specific applications may include in their representations links to other applications or services.
- *Uniform interface.* Our resources can be accessed in a uniform way using the HTTP interface.

Additionally, the HTTP protocol provides established and widely supported mechanisms for negotiating content and caching. Content negotiation allows the user agents to acquire user interface representations that can be rendered using the local look and feel. Caching helps to reduce the network bandwidth requirements by not sending full responses when the user agent already has up to date representations.

Our system relies on the HTTP-based change propagation mechanism to synchronize the state of the user interface on the user agent and the application. To ensure that they share a consistent state, we rely on TCP to deliver the requests in an orderly fashion, and HTTP pipelining when TCP connections are reused. The result is a system that provides eventual consistency. For example, in the example Texas Hold'em application, the initialization and showdown may take several seconds to complete when the application runs on the handheld device but when the processing of the request burst is completed the state on both devices is consistent.

In general, the eventual consistency model achieved by our system is appropriate for interactive applications that can tolerate delays. However, it should be noted that the synchronization performance varies depending on the hardware capabilities of the devices used and the network distance between them. Application developers can use the profiling tool provided with the toolkit to identify bottlenecks and adjust the complexity of the user interfaces for the less capable devices, or to accommodate the expected network delays in the intended usage environment.

Security and privacy have not been addressed in this paper. However, we are investigating the use of oAuth (Hammer-Lahav 2010) to enable authorization of user agents and HTTPS for providing confidentiality protection for the communication between user agents and applications.

In this chapter, we presented our initial experiences on using REST for scalable and adaptive sharing of native desktop and mobile applications. Although our efforts are currently limited to Qt applications, the results are encouraging and we plan to enable using standard web browsers as user agents. However, these are just firsts steps towards our vision that applications can be experienced from any remote devices. More research and prototyping is needed before the vision is fully realized.

Acknowledgements We would like to thank Saku Tiainen for developing the Texas Hold'em application used for testing, and the referee who provided feedback and suggestions that improved the content of this manuscript. The research was completed in the Cloud Software Program[4] of the Strategic Centre for Science, Technology and Innovation in the Field of ICT (TIVIT),[5] and was partially funded by the Finnish Funding Agency for Technology and Innovation (TEKES).[6]

[4]http://www.cloudsoftwareprogram.org.

[5]http://www.tivit.fi.

[6]http://www.tekes.fi.

References

Allamaraju, S.: RESTful Web Services Cookbook, pp. 90–91. O'Rilley Media, Sebastopol, California (2010)

Crockford, D.: The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627, IETF (2006). Http://www.ietf.org/rfc/rfc4627.txt

DeRose, S., Orchard, D., Maler, E.: XML linking language (XLink) version 1.0. W3C recommendation, W3C (2001). Http://www.w3.org/TR/2001/REC-xlink-20010627/

Gajos, K., Weld, D.S.: Supple: automatically generating user interfaces. In: IUI '04: Proceedings of the 9th International Conference on Intelligent User Interfaces, pp. 93–100. ACM, New York, NY, USA (2004)

Hammer-Lahav, E.: The OAuth 1.0 Protocol. RFC 5849 (informational), IETF (2010). Http://www.ietf.org/rfc/rfc5849.txt

Helms, J., Schaefer, R., Luyten, K., Vanderdonckt, J., Vermeulen, J., Abrams, M.: User interface markup language (UIML) version 4.0. Committee draft, OASIS (2008). Http://www.oasis-open.org/committees/download.php/28457/uiml-4.0-cd01.pdf

Khare, R., Taylor, R.N.: Extending the representational state transfer (rest) architectural style for decentralized systems. In: ICSE '04: Proceedings of the 26th International Conference on Software Engineering, pp. 428–437. IEEE Computer Society, Washington, DC, USA (2004)

Krasner, G.E., Pope, S.T.: A cookbook for using the model-view controller user interface paradigm in smalltalk-80. J. Object Oriented Program. **1**(3), 26–49 (1988)

Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., Lpez-Jaquero, V.: Usixml: a language supporting multi-path development of user interfaces. In: R. Bastide, P. Palanque, J. Roth (eds.) Engineering Human Computer Interaction and Interactive Systems, *Lecture Notes in Computer Science*, vol. 3425, pp. 200–220. Springer, Berlin, Heidelberg, New York (2005)

Nottingham, M.: Web Linking. RFC 5988 (proposed standard), IETF (2010). Http://www.ietf.org/rfc/rfc5988.txt

Richardson, L., Ruby, S.: RESTful Web Services, pp. 79–105. O'Rilley Media, Sebastopol, California (2007)

Roach, A.: A SIP Event Package for Subscribing to Changes to an HTTP Resource. RFC 5989 (proposed standard), IETF (2010). Http://www.ietf.org/rfc/rfc5989.txt

Stirbu, V.: A restful architecture for adaptive and multi-device application sharing. In: WS-REST '10: Proceedings of the First International Workshop on RESTful Design, pp. 62–66. ACM, New York, NY, USA (2010)

Stirbu, V., Leppanen, T.: An open platform for distributed, scalable and adaptive interactive applications for CE devices. In: The 8th Annual IEEE Consumer Communications and Networking Conference – Demos (CCNC'2011 – Demos). Las Vegas, NV, USA (2011)

Want, R., Pering, T.: System challenges for ubiquitous & pervasive computing. In: Proceedings of the 27th International Conference on Software Engineering, ICSE '05, pp. 9–14. ACM, New York, NY, USA (2005)

Chapter 9
RESTful Service Development for Resource-Constrained Environments

Amirhosein Taherkordi, Frank Eliassen, Daniel Romero, and Romain Rouvoy

Abstract The use of resource-constrained devices, such as smartphones, PDAs, Tablet PCs, and *Wireless Sensor Networks* (WSNs) is spreading rapidly in the business community and our daily life. Accessing services from such devices is very common in ubiquitous environments, but mechanisms to describe, implement and distribute these services remain a major challenge. *Web services* have been characterized as an efficient and widely-adopted approach to overcome heterogeneity, while this technology is still heavyweight for resource-constrained devices. The emergence of REST architectural style as a lightweight and simple interaction model has encouraged researchers to study the feasibility of exploiting REST principles to design and integrate services hosted on devices with limited capabilities. In this chapter, we discuss the state-of-the-art in applying REST concepts to develop Web services for WSNs and smartphones as two representative resource-constrained platforms, and then we provide a comprehensive survey of existing solutions in this area. In this context, we report on the DIGIHOME platform, a home monitoring middleware solution, which enables efficient service integration in ubiquitous environments using REST architectural style. In particular, we target our reference platforms for home monitoring systems, namely WSNs and smartphones, and report our experiments in applying the concept of *Component-Based Software Engineering* (CBSE) in order to provide resource-efficient RESTful distribution of Web services for those platforms.

Introduction

Pervasive environments support context-aware applications that adapt their behavior by reasoning dynamically over the user and the surrounding information. This contextual information generally comes from diverse and heterogeneous entities,

A. Taherkordi (✉)
Department of Informatics, University of Oslo, PO Box 1080 Blindern, 0316 Oslo, Norway
e-mail: amirhost@ifi.uio.no

such as physical devices, *Wireless Sensor Networks* (WSNs), and smartphones. In order to exploit the information provided by these entities, a middleware solution is required to collect, process, and distribute the contextual information efficiently. However, the heterogeneity of systems in terms of technology capabilities and communication protocols, the mobility of the different interacting entities, and the identification of adaptation situations make this integration difficult. Thus, this challenge requires a flexible solution in terms of communication support and context processing to leverage context-aware applications on the integration of heterogeneous context providers.

In particular, a solution dealing with context information and control environments must be able to connect with a wide range of device types. However, the resource scarceness in WSNs and mobile devices makes the development of such a solution very challenging. In this chapter, we propose the DIGIHOME platform, a simple but efficient service-oriented middleware solution to facilitate context-awareness in pervasive environments. Specifically, DIGIHOME provides support for the *integration*, *processing* and *adaptation* of the context-aware applications. Our solution enables the integration of heterogeneous computational entities by relying on the *Service Component Architecture* (SCA) model (Open SOA 2007), the REST (*REpresentational State Transfer*) principles (Fielding 2000), standard discovery and communication protocols, and resource representation formats. We combined SCA and REST in our solution in order to foster reuse and low coupling between the different services that compose the platform. We believe that the REST concepts of simplicity (in terms of interaction protocols) and flexibility (regarding the supported representation formats) make it a suitable architecture style for pervasive environments. The DIGIHOME platform presented in this chapter is an improved version of our work introduced in Romero et al. (2010a).

The remainder of this chapter is organized as follows. We start by reporting on the existing RESTful solutions for constrained devices (cf. RESTful Solutions for Constrained Platforms). Then, we describe a smart home scenario in which we identify the key challenges in pervasive environments that motivate this work (cf. RESTful Integration of Services: A Home Monitoring Scenario). We continue by the description of DIGIHOME, our middleware platform to support the integration of systems-of-systems in pervasive environments (cf. The DIGIHOME Service-Oriented Platform). Then, we discuss the benefits of our approach as well as future opportunities (cf. Future: Horizons and Challenges) before concluding (cf. Conclusion).

RESTful Solutions for Constrained Platforms

For years, the use of REST in mobile devices was restricted to client-side interaction from web browsers. As consequence of Moore's law, the computing capabilities of mobile devices are quickly increasing. In particular, we observe that mobile devices move from simple service consumers to service providers. As an example, several activities have been recently initiated in order to enable the deployment of a web server within a mobile device (Wikman and Dosa 2006; Nokia 2008; Pham and Gehlen 2005; Srirama et al. 2006; The Apache Software Foundation 2009). Another example is the OSGi initiative, which includes a HTTPd bundle in most of its platforms (OSGi Alliance 2009, 2007). The strength of these mobile web servers is that the information they publish can be dynamically tuned depending on the context surrounding the mobile device.

From Supporting Lightweight Web Services. Recently, mobile phone platforms have drawn a lot of attention from manufacturers and users, mainly due to the opportunity to embark what has become lightweight computers in everyone's pocket. As such, smartphones are now equipped with GPS, high speed cellular network access, wireless network interfaces, and various sensors (accelerometers, magnetometers, etc.). However, phone manufacturers keep offering platforms that differ radically in their operating system and API choices.

Android is an example of operating system for these mobile devices that includes middleware and key applications based on a tailored version of the Linux kernel. The Android platform allows developers to write application code in the Java language and to control the device via Google-based Java libraries. According to *The Nielsen Company*, unit sales for Android OS smartphones ranked first among all smartphone OS handsets sold in the U.S. during the first half of 2010. These numbers confirm the increasing success of smartphones in the population, and in the context of the DigiHome platform, we benefit from this acceptation to improve future applications by proposing a service-oriented platform exploiting smartphones and user usages.

Lightweight Web Services have been motivated by the concept of the Internet of Things – a technological revolution to connect daily objects and devices to large databases and networks, and therefore to the Internet. In this model, Web services standards are used to integrate WSNs and the Internet, e.g., in SOCRADES (de Souza et al. 2008), Web services are tailored at the gateway device where the *Device Profile for Web Services* (DPSW) is used to enable messaging, discovery, and eventing on devices with resource restrictions. However, in the context of WSN, since the current footprint of DPSW for sensor nodes is too large, this solution is only deployable on gateways. To overcome this issue, Priyantha et al. (2008) propose SOAP-based Web services, called *Tiny web services*, for WSNs. However, apart from its complexity, this work mainly focuses on low-level issues related to Web integration in TinyOS-based sensor networks.

To Providing RESTful Middleware Solutions. To the best of our knowledge, the number of middleware solutions dealing with the support of RESTful services in constrained devices is very limited. In fact, these approaches are proposed due to the high resource needs and complexity of SOAP-based Web Service protocols.

The Restlet Framework[1] is the first RESTful web framework for Java developers that targets Android for deployment on compatible smartphones. Restlet's vision is that the Web is becoming ubiquitous and that REST, as the architecture style of the Web, helps developers to leverage all HTTP features. In particular, Restlet on Android supports both client-side and server-side HTTP connectors. However, Restlet does not include support for the discovery of RESTful services, which is a fundamental requirement in pervasive environments.

In WSNs, TinyREST is one of the first attempts to integrate WSNs into the Internet (Luckenbach et al. 2005). It uses the HTTP-based REST architecture to retrieve/update the state of sensors/actuators. The TinyREST gateway maps a set of HTTP requests to TinyOS messages in order to link MICA motes (Hill and Culler 2002) to any Internet client. Beside the fact that in TinyREST only a gateway is able to connect to the Internet (not any individual sensor node), this approach fails to follow all standard HTTP methods. The work reported in Guinard et al. (2009) also presents a REST-based gateway to bridge the Web requests to powerful SunSPOT nodes.

RESTful Integration of Services: A Home Monitoring Scenario

In this section, we report on a smart home scenario to clarify the motivations of our work. A smart home generally refers to a house environment equipped with several types of computing entities, such as *sensors*, which collect physical information (temperature, movement detection, noise level, light, etc.), and *actuators*, which change the state of the environment. In this scenario, we consider a smart home equipped with occupancy, smoke detection, and temperature sensors. These tiny devices have the ability to collect context information and to communicate wirelessly with each other, in order to identify the context situation of the environment. In addition to that, we can also use actuators to physically control lights, TV, and air conditioning. Figure 9.1 illustrates the integration of these sensors and actuators in our scenario. As appreciated in this figure, the different entities use heterogeneous protocols to interact. In the scenario, the smartphones provide information about the user preferences for the home configuration. Conflicts between the user preferences are resolved by giving priority to the person who arrived first to the room. The mobile devices also have an application that enables the control of the actuators present in the different rooms. This application can be adapted when there are changes in the actuator's configuration. Finally, there is a *Set-Top Box* (STB) which is able to gather information, and interact with the other co-located devices.

[1]Restlet:`http://www.restlet.org`

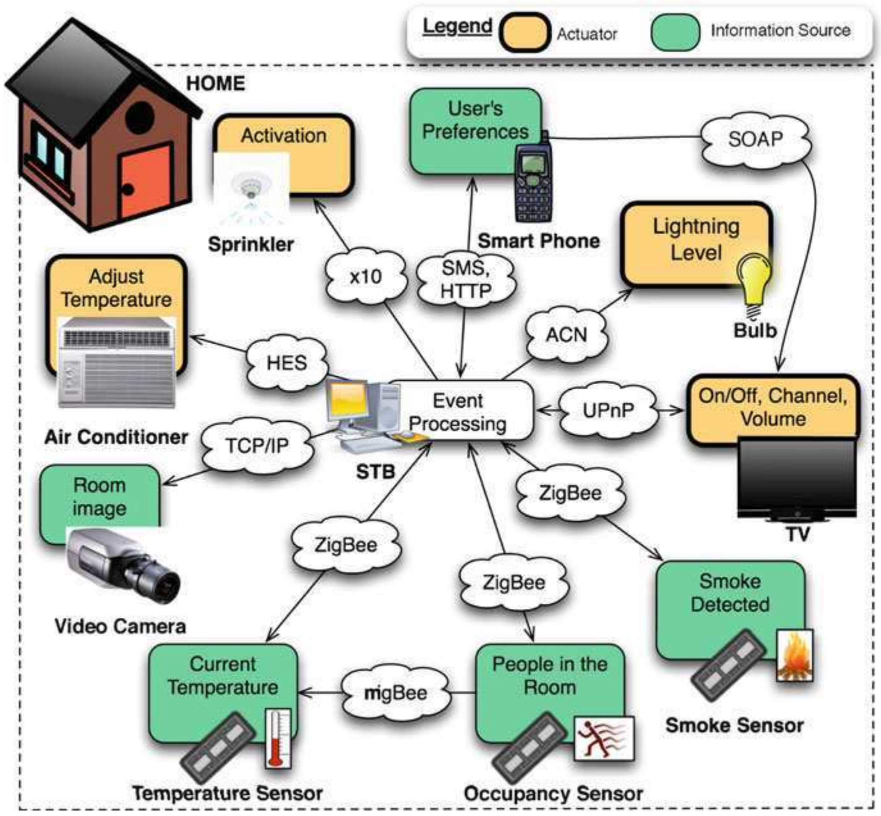


Fig. 9.1 Interactions between the smart home devices

To show how the different elements of our scenario interact, we present three different situations:

Situation 1: Alice arrives to the living room. The occupancy sensor detects her presence and triggers the temperature sensors to decrease the sampling rate of data. It also notifies the STB that the room is occupied by somebody, which in turn tries to identify the occupant by looking for a profile in her mobile device. When Alice's profile is found, the STB loads it and adjusts the temperature and lightening level of the room according to Alice's preferences.

Situation 2: The sensors detect smoke and notify the STB, which in turn uses the occupancy sensor and realizes that the house is empty. The STB therefore sends an SMS to Alice, including a picture of the room captured using the surveillance camera. After checking the picture, Alice decides to remotely trigger the sprinklers using her mobile device. She also tells the system to alert the fire department about the problem. If Alice does not reply to the STB within 5 min, the system activates automatically the sprinklers and alerts the fire department.

Situation 3: Alice installs a new TV in the bedroom. The STB detects the presence of the new device, identifies it, and downloads the corresponding control software from an Internet repository. The platform tries to locate the available mobile devices, using a discovery protocol, and finds Alice's mobile device. The STB proposes to update the mobile device with the components for controlling the new TV.

Key Challenges

The various situations we described above allow us to identify several key challenges in terms of:

1. *Integration of multi-scale entities*: The mobile devices and sensors have different hardware and software capabilities, which make some devices more powerful than others. Therefore, the integration of these entities requires a flexible and simple solution that supports multiple interaction mechanisms and considers the restricted capabilities of some devices. In particular, regarding sensor nodes, the immaturity of high-level communication protocols, as well as the inherent resource scarceness, bring two critical challenges to our work: (1) how to connect sensor nodes to mobile devices and actuators through a standard high-level communication protocol and (2) the framework which runs over sensor nodes for supporting context-awareness and adaptation should not impose high resource demands.
2. *Entity mobility*: In our scenario, computational entities appear and disappear constantly. In particular, mobile devices providing user profiles are not always accessible (they can be turned off or the owner can leave the house with them). In a similar way, the actuators can be replaced or new ones can be added. Thus, we need to discover new entities dynamically as well as to support device disconnections.
3. *Information processing and adaptation*: In order to support adaptation, we first need to identify the situations in which the adaptation is required. We have a lot of information that is generated by the different devices in the environment. Therefore, we need to define which part of this information is useful to identify relevant situations and react accordingly. In our scenario, those situations include the load of Alice's profile and the adjustment of the temperature, the sending of an alert via SMS in case of an emergency, and the adaptation of Alice's mobile device to control the new TV in her bedroom.

The DIGIHOME Service-Oriented Platform

The integration, mobility and adaptation issues impose several requirements for the development of smart home environments. To deal with these issues, we propose a comprehensive and simple solution called DIGIHOME, which leverages on the

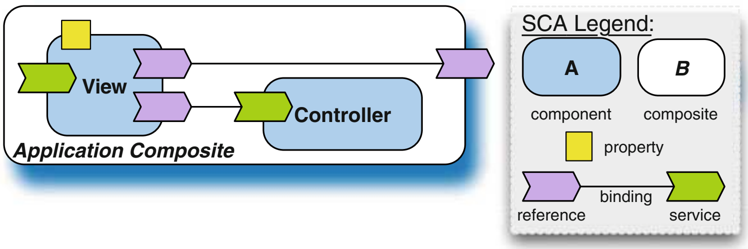
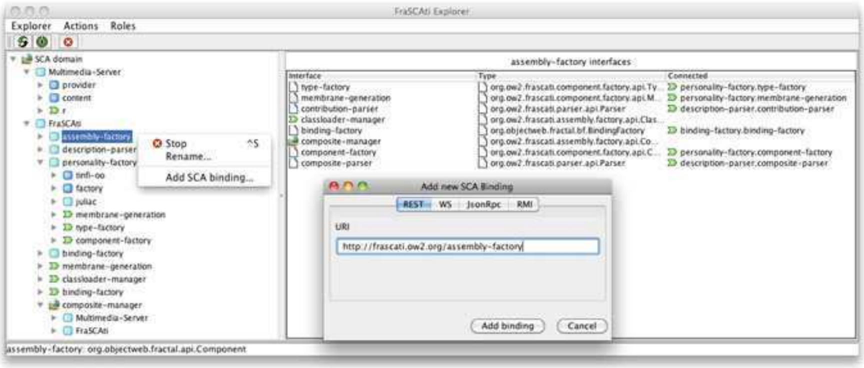


Fig. 9.2 Overview of the SCA component model

integration of events and context information as well as the dynamic configuration of applications by using the REST architectural style. In particular, we propose a flexible architecture that modularizes the different concerns associated with event processing in ubiquitous environments by applying existing standards and approaches. In our solution, we support the integration of various event sources (e.g., sensors in our scenario), context providers (e.g., mobile devices), and other kind of services (e.g., actuators and reconfiguration services) implemented with a variety of technologies and interacting via different protocols. Indeed, DigiHome deals with protocol heterogeneity by enabling the runtime incorporation of different communication mechanisms when required thanks to the SCA isolation of non-functional concerns.

Background on SCA and FraSCAti

The *Service Component Architecture* (SCA) (Beisiegel et al. 2007) is a set of specifications for building distributed applications based on *Service-Oriented Architecture* (SOA) and *Component-Based Software Engineering* (CBSE) principles. As illustrated in Fig. 9.2, the basic construction blocks of SCA are software *components*, which have *services* (or provided interfaces), *references* (or required interfaces) and expose *properties*. The references and services are connected by means of *wires*. SCA specifies a hierarchical component model, which means that components can be implemented either by primitive language entities or by subcomponents. In the latter case the components are called *composites*. SCA is designed to be independent from programming languages, *Interface Definition Languages* (IDL), communication protocols, and non-functional properties. In particular, to support interaction via different communication protocols, SCA provides the notion of *binding*. For SCA references, bindings describe the access mechanism used to invoke a service. In the case of services, the bindings describe the access mechanism that clients use to execute the service.



assembly-factory interfaces

Interface	Type	Connected
type-factory	org.ow2.frascati.component.factory.api.Ty...	personality-factory.type-factory
membrane-generation	org.ow2.frascati.component.factory.api.M...	personality-factory.membrane-generation
contribution-parser	org.ow2.frascati.parser.api.Parser	description-parser.contribution-parser
classloader-manager	org.ow2.frascati.assembly.factory.api.Clas...	
binding-factory	org.objectweb.fractal.bf.BindingFactory	binding-factory.binding-factory
composite-manager	org.ow2.frascati.assembly.factory.api.Co...	
component-factory	org.ow2.frascati.component.factory.api.C...	personality-factory.component-factory
composite-parser	org.ow2.frascati.parser.api.Parser	description-parser.composite-parser

http://frascati.ow2.org/assembly-factory

assembly-factory: org.objectweb.fractal.api.Component

Fig. 9.3 Screenshot of FRASCATI EXPLORER

Listing 9.1 reflects part of the configuration depicted in Fig. 9.2 using the SCA assembly language:

```
<composite name="MyApp" xmlns="http://www.osoa.org/xmlns/sca/1.0">
  <service name="run" promote="View/run"/>
  <component name="View">
    <implementation.java class="app.gui.SwingGuiImpl"/>
    <service name="run">
      <interface.java interface="java.lang.Runnable"/>
    </service>
    <reference name="model" autowire="true">
      <interface.java interface="app.ModelService"/>
    </reference>
    <property name="orientation">landscape</property>
  </component>
  <!-- ... -->
</composite>
```

Listing 9.1 Description of the application MyApp

The FRASCATI middleware platform focuses on the development and execution of SCA-based distributed applications (Seinturier et al. 2009). The platform itself is built as an SCA application – i.e., its different subsystems are implemented as SCA components. FRASCATI extends the SCA component model to add reflective capabilities in the application level as well as in the platform. In particular, Fig. 9.3 illustrates the FRASCATI EXPLORER toolset, which provides the capacity of introspecting and reconfiguring an SCA application and FRASCATI interactively. These reconfigurations can also be automated through the use of reconfiguration scripts based on the FSCRIPT syntax (David et al. 2009).

Furthermore, the FRASCATI platform applies interception techniques for extending SCA components with non-functional services, such as confidentiality, integrity, and authentication. In this way, FRASCATI provides a flexible and extensible component model that can be used in distributed environments to deal with heterogeneity. In our context, as later reported in this section, we benefit from the protocol independence of SCA to define REST-oriented bindings that

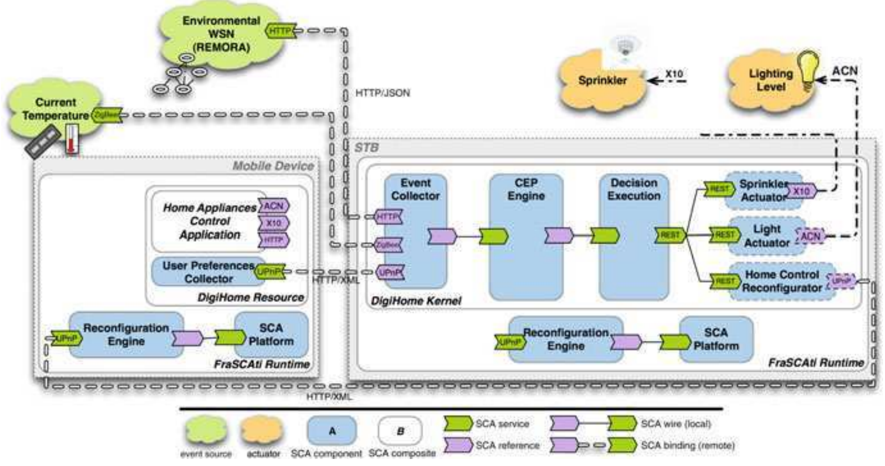


Fig. 9.4 Overview of the DIGIHOME RESTful architecture

provide a simple and flexible mechanism for tracking activities of mobile users as well as XQuery component implementations, which is not provided by the standard FRASCATI distribution. Furthermore, the FRASCATI capabilities in terms of runtime adaptation for applications and the platform itself, make it a suitable option for customizing the DIGIHOME platform whenever required.

DigiHome: An Example of RESTful Architecture

In DIGIHOME, we follow the REST principles (Fielding 2000) to reduce the coupling between entities by focusing on data exchange interactions, which can have multiple representations (e.g., XML and JSON). In a similar way, for supporting the integration of devices with restricted capabilities, DIGIHOME promotes the usage of a lightweight API and simple communication protocols as stated by REST. In particular, our solution benefits from WSNs in order to process simple events and make local decisions when possible, by means of the REMORA component model (Taherkordi et al. 2010), which is a component model for WSNs based on SCA. Finally, the platform uses a *Complex Event Processing* (CEP) engine for event processing and the adaptation of applications and room configuration. Figure 9.4 depicts the general architecture of the platform, and in the rest of the section we provide a detailed description of the different elements of the platform.

DIGIHOME Kernel. The kernel of the platform modularizes the main responsibilities for home monitoring. This means that the kernel contains the functionalities required for event collecting, event processing, and deciding and executing the required adaptations of the applications deployed on DIGIHOME resources as well as the room configurations. In DIGIHOME, the Event Collector retrieves and stores the recent information produced by event and context sources, such as sensors and mobile devices. The CEP Engine is responsible for event processing and uses the Decision Executor to perform actions specified by the Adaptation Rules (defined in the CEP Engine). Following a plug-in mechanism, the different Actuator components grant access to the available actuator services in the environments. This means that the different actuators are optional, deployed according to the current service configuration and deployed on different devices.

To enable the communication between different clients and to support the mobility of services and mobile devices, we also incorporate ubiquitous bindings in SCA (Romero et al. 2010b). These bindings bring into SCA existing discovery protocols, such as UPnP (UPnP Forum 2008) and SLP (Guttman et al. 1999), providing the possibility to establish spontaneous communications. Furthermore, the ubiquitous bindings improve the context information advertisements with *Quality of Context* (QoC) (Krause and Hochstatter 2005) attributes for provider selection. Once the services are discovered, the ubiquitous bindings are flexible enough to allow the interaction via standard bindings, such as REST. The use of these ubiquitous bindings, as well as the modularization of the different concerns, makes it easy to distribute the different responsibilities in DIGIHOME.

DIGIHOME Resources. A DIGIHOME Resource is an SCA component providing and/or consuming events to/from other DIGIHOME Resources. In our scenario, the mobile device executes a DIGIHOME Resource that offers the user preferences as context information and hosts an adaptive application enabling the control of home appliances (that also consumes events indirectly in order to be adapted). The DIGIHOME Kernel can also be considered as a DIGIHOME Resource. Because our solution is based in standards, and in hiding the service implementation with SCA, we can easily integrate other services in the smart home that are not part of the infrastructure (in particular, the actuators). In a similar way, we are exposing the DIGIHOME Resources via ubiquitous bindings so that other applications (that are not part of DigiHome) can benefit from the services offered by the platform.

Listing 9.2 reports on the SCA assembly descriptor of the LightActuator component we developed for interacting with the X10-compliant light appliance using the REST architectural style (as described in Fig. 9.4):

```
<composite name="DigiHome.Kernel" xmlns="http://www.osoa.org/xmlns/sca/1.0">
 <component name="LightActuator">
   <implementation.java class="digihome.LightActuatorImpl"/>
   <service name="actuator"
            xmlns:rest="http://frascati.ow2.org/xmlns/rest/1.0">
     <rest:interface.wadl description="DigiHome" resource="LightResource"/>
     <rest:binding.http uri="/light"/>
   </service>
   <reference name="light">
     <interface.java interface="digihome.ILightActuator"/>
     <home:binding.x10 xmlns:home="http://frascati.ow2.org/xmlns/home/1.0"/>
   </reference>
 </component>
</composite>
```

Listing 9.2 Description of the *LightActuator* resource

This DigiHome *Resource* is developed in Java (line 3) and exposes the service it provides as a REST resource (lines 4–8). Technically, we extended the FraSCAti platform to support the REST architectural style (Fielding 2000). This extension includes the support for the *Web Application Description Language* (WADL) standard (W3C 2009) as a new interface type used to describe the resource `actuator` and for the HTTP communication protocol as a new binding type to access this resource. The REST bindings support multiple context representations (e.g., XML, JSON, and Java Object Serialization) and communication protocols (HTTP, XMPP, FTP, etc.). This flexibility allows us to deal with the heterogeneous context managers and context-aware applications as well as with the different capabilities of the devices that execute them. Details about the architecture of these bindings are further presented in Romero et al. (2009).

DigiHome CEP Engine. To manage the events in our scenario, we need a decision-making engine that can process them and create relations to identify special situations, using predefined rules. In order to identify the desired events, the CEP Engine requires to communicate with an Event Collector, which is in charge of dealing with the subscriptions to the event sources. When an adaptation situation is detected, a corresponding action is triggered, which can go from an instruction to an actuator, to the adaptation of the system by adding or removing functionality. These actions are received by the Decision Executor, which has the responsibility of communicating with the different actuators in the environment.

In DigiHome, for the event processing in the set-top box, we use Esper (EsperTech 2009), a Java open source stream event processing engine, to deal with the event management and decision making process. We chose Esper for our platform because it is the most supported open source project for CEP and is very stable, efficient, and fairly easy to use. The following code excerpt shows an example of an Esper rule used in our scenario:

```
select sum(movement)
  from MovementSensorEvent.win:time(60 sec)
```

In this rule, we can see the use of a time window, which is a moving interval of time. The rule collects all the events from the movement sensor from the last 60 s. By doing this, we can know if a user is still in the room or has already left, and adapt the room accordingly.

DigiHome Intermediaries. REST enables *Web Intermediaries* (WBI) to exploit the requests exchanged by the participants in the communication process. WBI are computational entities that are positioned between interacting entities on a network to tailor, customize, personalize, or enhance data as they flow along the stream (IBM 2009). Therefore, we can benefit from this opportunity to improve the performance of DigiHome. When the provided context information does not change much in time, the messages containing this information can be marked as cacheable within the communication protocol. This kind of annotation enables WBI caches to quickly analyze and intercept context requests always returning the

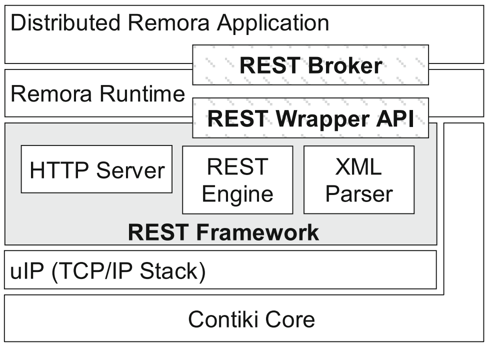


Fig. 9.5 Architecture of RESThing framework

same document. A similar optimization applies to security issues and the filtering of context requests. Indeed, by using proxy servers as WBI, we can control the requested context resources and decide whether the incoming (or outgoing) context requests need to be propagated to the web server publishing the context resource. Other kinds of WBI can also be integrated in the system to operate, for example, resource transcoding, enrichment or encryption.

DIGIHOME *Wireless Sensor Network*

In order to consume events from WSNs, we use the REMORA Component Framework (Taherkordi et al. 2010). This framework is an extension of SCA that brings component-based development into WSNs. REMORA proposes a RESTful mechanism to exchange events, which is encapsulated in an SCA component. We reuse this mechanism in order to define DIGIHOME resources for WSNs (so called REMORA RESOURCES), which are able to produce and consume simple objects in the DIGIHOME platform. With these resources, we improve the efficiency of the system because the WSN is able to process simple events instead of going through the DIGIHOME KERNEL for making decisions. The core of our framework enables in-WSN decisions, whenever an event should be processed with other relevant events generated by other sensor nodes. As an example, when a temperature sensor detects a high temperature, to know if there is a fire, it needs to become aware of the smoke density in the room – i.e., communicating with the smoke detecting sensors. Furthermore, benefiting from the DIGIHOME modularization of concerns, as well as the transparent communication promoted by SCA, DIGIHOME objects can consume/notify events from/to REMORA RESOURCES with a small effort.

This framework presents an IP-based sensor network system where nodes can directly integrate to modern IT systems through RESTful Web services. This approach relies on the IP protocol stack implemented in Contiki operating system. Contiki has made a considerable effort on the provision of IPv4 and IPv6 protocols on the common types of sensor nodes with constrained resources. Software architecture of RESThing is shown in Fig. 9.5. It consists of HTTP Server, REST Engine, SAX based XML parser and Logger modules. RESThing offers an interface to create resources since they are the main abstractions of RESTful Web services.

The REMORA runtime is integrated with the REST framework through the *REST Wrapper API*. Wrapper API is one of the main features of REMORA, providing a well-described method for integrating a REMORA application with underlying system software (Taherkordi et al. 2010). *REST Broker* contains a set of REMORA components processing REST requests received from REMORA runtime. Specifically, it is an intermediate module for handling the REST requests received from a Web client or sent from the sensor node to a node hosting RESTful Web services. The broker is also in charge of retaining the list of application-specific resources and the corresponding REMORA Web services APIs.

HTTP server is a small footprinted server to handle the incoming and outgoing HTTP requests. It provides interface to perform certain HTTP related tasks such as accessing request details (headers, entity body and URL path), constructing an HTTP response, etc. Both REST Engine and SOAP Engine work on top of the HTTP server. The REST framework also includes a XML parser to parse requests in XML format. A simple XML parser developed by a third-party (simplexml parser) is ported to Contiki for this purpose. It is very small in code size and being a non-validating SAX based parser makes it memory efficient. A minimal SOAP processing engine is also provided to fulfill SOAP-based Web service invocations. To do that, it reuses the HTTP server and XML parser components. The engine parses the SOAP message using the XML parser, extracts the method information and executes it, finally the response SOAP message is built using the XML parser.

Future: Horizons and Challenges

Applications and technologies for the Internet of Things are still in the promotional phase of design and development. There are numerous hurdles against large-scale use of the Internet of Things originated from the lack of standards and unmature business models. We believe that the primary IoT concern in the future will be on the integration of large-scale enterprise systems with resource-constrained platforms. The future WSNs and RFID systems, for example, can trigger business processes, adversely; actions can be triggered on them by a business process being executed on the Internet. When such platforms are involved in the business process lifecycle, in addition to today's communication and distribution issues, several new challenges arise that are enterprise-specific and difficult to be addressed on resource-limited platforms, such as *workflow* management systems. As a result, customizing the heavyweight enterprise software infrastructures for embedded and resource-poor systems is envisaged to be a potential research trend of IoT in the future. On the future horizons of IoT, internet services also serve a key role. The number of embedded and tiny devices integrated to future IoT will be dramatically increased so that distributed applications deployed over infrastructures that may encompass tens of thousands of tiny devices, where each device exhibits a high number of services. Strategies to locate services, as well as devices hosting the services could be a crucial challenge in the future IoT. It is required to shift the thoughts from *things* in the Internet to *services*, where applications deal with virtual things able to scale up to the plentiful services over the Internet. These virtual platforms offer a new higher level of abstraction that hides the low level real things and represent a different set of things which are characterized as the virtual and new members of IoT. The way to compose low level real services and expose them in the *virtual things* level could be a challenging area in the future IoT.

Conclusion

In this chapter, we have presented DIGIHOME, a platform addressing the mobility, heterogeneity, and adaptation of smart entities. In particular, DIGIHOME detects adaptation situations by integrating context information using an SCA-based architecture. This architecture promotes the modularization of concerns and fosters the application of the REST principles by exploiting the SCA extensibility. The simplicity and data orientation of REST, combined with the SCA independence of implementation technologies, make DIGIHOME an attractive solution to deal with heterogeneity in terms of interactions. The definition and application of ubiquitous bindings in the platform enable spontaneous communication by means of standard protocols (e.g., UPnP and SLP), and furnish context provider selection (based on QoC attributes). On the other hand, the modularized architecture of DIGIHOME allows the definition of variants for the platform, called DIGIHOME resources, that can be deployed on resource-constrained devices. The functionality of these resources is exposed as services, accessible via several protocols, which can be accessed by clients that do not have to be part of the platform. Furthermore, the clear separation of concerns in the DIGIHOME architecture encourages the exploitation of WSNs for simple processing and local decision making. The suitability of our platform for context integration was evaluated with different discovery and context representations.

References

simplexml parser http://simplexml.sourceforge.net, 2009.

M. Beisiegel et al. Service Component Architecture. http://www.osoa.org, 2007.

P.-C. David, T. Ledoux, M. Léger, and T. Coupaye. FPath and FScript: language support for navigation and reliable reconfiguration of FRACTAL architectures. *Annales des Télécommunications*, 64(1–2): 45–63, January 2009.

L. de Souza, P. Spiess, D. Guinard, M. Khler, S. Karnouskos, and D. Savio. SOCRADES: a web service based shop floor integration infrastructure. In *The Internet of Things*, volume 4952 of *LNCS*, pages 50–67. Springer, Berlin, Heidelberg, New York, 2008.

EsperTech. Esper. http://esper.codehaus.org, 2009.

R. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD Thesis. University of California, Irvine, USA, 2000.

D. Guinard, V. Trifa, T. Pham, and O. Liechti. Towards physical mashups in the web of things. In *INSS'09: Proceedings of the 6th International Conference on Networked Sensing Systems*, pages 196–199, IEEE, Pittsburgh, PA, USA, 2009.

E. Guttman, C. Perkins, J. Veizades, and M. Day. Service Location Protocol, Version 2. RFC 2608 (Proposed Standard). http://tools.ietf.org/html/rfc2608, June 1999.

J. L. Hill and D. E. Culler. Mica: a wireless platform for deeply embedded networks. *IEEE Micro*, 22: 12–24, 2002.

IBM. Web Intermediaries (WIB). http://www.almaden.ibm.com/cs/wbi, 2009.

M. Krause and I. Hochstatter. Challenges in modelling and using quality of context (QoC). In *Proceedings of the 2nd International Workshop on Mobility Aware Technologies and Applications*, pages 324–333, Montreal, Canada, 2005.

T. Luckenbach, P. Gober, K. Kotsopoulos, A. Kim, and S. Arbanowski. TinyREST: a protocol for integrating sensor networks into the internet. In *REALWSN'05: Proceedings of the Workshop on Real-World WSNs*, Stockholm, Sweden, 2005.

Nokia. Mobile Web Server. http://wiki.opensource.nokia.com/projects/Mobile_Web_Server, 2008.

Open SOA. Service Component Architecture Specifications, 2007.

OSGi Alliance. OSGi – The Dynamic Module System for Java. http://www.osgi.org, 2009.

OSGi Alliance. About the Osgi Service Platform – Technical Whitepaper Revision 4.1. http://www.osgi.org/documents, 2007.

L. Pham and G. Gehlen. Realization and performance analysis of a SOAP server for mobile devices. In *Proceedings of the 11th European Wireless Conference*, volume 2, pages 791–797, VDE Verlag, Nicosia, Cyprus, April 2005.

N. B. Priyantha, A. Kansal, M. Goraczko, and F. Zhao. Tiny Web Services: design and implementation of interoperable and evolvable sensor networks. In *SenSys'08: Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, pages 253–266, ACM, Raleigh, NC, USA, 2008.

D. Romero, G. Hermosillo, A. Taherkordi, R. Nzekwa, R. Rouvoy, and F. Eliassen. RESTful integration of heterogeneous devices in pervasive environments. In *Proceedings of the 10th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'10)*, volume 6115 of *LNCS*, pages 1–14. Springer, Berlin, Heidelberg, New York, June 2010.

D. Romero, R. Rouvoy, L. Seinturier, and P. Carton. Service discovery in ubiquitous feedback control loops. In *Proceedings of the 10th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'10)*, volume 6115 of *LNCS*, pages 113–126. Springer, Berlin, Heidelberg, New York, June 2010.

D. Romero, R. Rouvoy, L. Seinturier, S. Chabridon, C. Denis, and P. Nicolas. Enabling context-aware web services: a middleware approach for ubiquitous environments. In Michael Sheng, Jian Yu, and Schahram Dustdar, editors, *Enabling Context-Aware Web Services: Methods, Architectures, and Technologies*. Chapman and Hall/CRC, London, 2009.

L. Seinturier, P. Merle, D. Fournier, N. Dolet, V. Schiavoni, and J. -B. Stefani. Reconfigurable sca applications with the frascati platform. In *SCC'09: Proceedings of the IEEE International Conference on Services Computing*, pages 268–275, IEEE Computer Society, Washington, DC, USA, September 2009.

S. N. Srirama, M. Jarke, and W. Prinz. Mobile web service provisioning. In *International Conference on Advanced International Conference on Telecommunications / Internet and Web Applications and Services*, IEEE, page 120, 2006.

A. Taherkordi, F. Loiret, A. Abdolrazaghi, R. Rouvoy, Q. Le Trung, and F. Eliassen. Programming sensor networks using REMORA component model. In *Proceedings of the 6th IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS'10)*, page 15, Santa Barbara, California, USA France, 06 2010.

The Apache Software Foundation. HTTP Server Project. http://httpd.apache.org, 2009.
UPnP Forum. UPnP Device Architecture 1.0. http://www.upnp.org/resources/documents.asp, 2008.
W3C. Web Application Description Language (WADL). https://wadl.dev.java.net, 2009.
J. Wikman and F. Dosa. Providing HTTP Access to Web Servers Running on Mobile Phones, 2006.

Chapter 10
A REST Framework for Dynamic Client Environments

Erik Albert and Sudarshan S. Chawathe

Abstract The REST Framework for Dynamic Client Environments (RFDE) is a method for building RESTful Web applications that fully exploit the diverse and rich feature-sets of modern client environments while retaining functionality in the absence of these features. For instance, we describe how an application may use a modern JavaScript library to enhance interactivity and end-user experience while also maintaining usability when the library is unavailable to the client (perhaps due to incompatible software). These methods form a framework that we have developed as part of our work on a Web application for presenting large volumes of scientific datasets to nonspecialists.

Introduction

The REST Framework for Dynamic Client Environments (RFDE) is a method for building RESTful Web applications (Fielding and Taylor 2002; Fielding 2000; Pautasso et al. 2008) that fully exploit the diverse and rich feature-sets of modern client environments while retaining functionality in the absence of these features. For instance, we describe how an application may use a modern JavaScript library to enhance interactivity and end-user experience while also maintaining usability when the library is unavailable to the client (perhaps due to incompatible software). These methods form a framework that we have developed as part of our work on a Web application for presenting large volumes of scientific datasets to nonspecialists.

The key problem addressed by the framework is: How do we build a robust and scalable Web application that, on one hand, uses to its advantage the numerous and increasingly capable clients and client-side libraries (e.g., Scriptaculous,

S.S. Chawathe (✉)
Department of Computer Science, University of Maine, 237 Neville Hall,
Orono, ME 04469-5752, USA
e-mail: chaw@cs.umaine.edu

OpenLayers) but, on the other hand, retains all important functionality when one or more such client features are unavailable? More specifically, how do we combine the benefits of the REST approach to Web application design with those of active client-side features such as JavaScript and techniques such as Ajax (Asynchronous JavaScript and XML) (Garrett 2005)?

To reach a wide audience, a Web application must be able to support a wide range of client capabilities. Some mobile clients and clients on older computers often cannot use the latest Web technologies such as Adobe Flash, scalable vector graphics (SVG) (Jackson and Northway 2005), Java applets, or even advanced JavaScript. In order to develop an application that is accessible to the largest audience, developers often design for a simple set of capabilities and eschew the newer technologies. Alternatively, developers utilize new technologies and provide an alternative, reduced-functionality version for clients that cannot support the chosen technologies. And very often, unfortunately, Web applications will simply display a requirements message to the reduced-capability clients and provide no functionality at all. The RFDE framework provides a much more attractive option, as it permits the use of modern JavaScript and other features while retaining usability on clients without these features, and permits the Web programmer to support all such clients without explicitly writing code to handle the many cases. In an RFDE Web application, requests from a client returns a version of the application that is best matched to that client's supported, and active, features. The RFDE framework also endows an application with the ability to automatically upgrade itself using JavaScript and Dynamic HTML (DHTML) to a representation that can take advantage of more dynamic and advanced client features when they are available.

In the remainder of this chapter, we will describe the Climate Data Explorer, a climatological web application that inspired the RFDE framework, and identify the types of applications that can benefit from this approach. We will then introduce widgets and application templates, which are the building blocks of an RFDE application, and describe how they can be designed to target a large number of client environments with varying capabilities. Next, we will describe how we represent and maintain the state of a dynamic and event-driven application that is implemented using a RESTful, stateless application server. Finally, we will describe some of the work related to the RFDE framework, summarize the approach, and describe some possible future enhancements.

Motivating Case Study: A Climate Data Explorer

We describe a concrete application, the *Climate Data Explorer* (henceforth, *CDX*), that motivates our design criteria and also serves as a running example for illustrating the RFDE framework in this chapter. The primary goal of CDX is enabling nonspecialists to intuitively and interactively explore an integrated view of a large and diverse collection of datasets related to climate, with emphasis on the spatial and temporal attributes of this data.

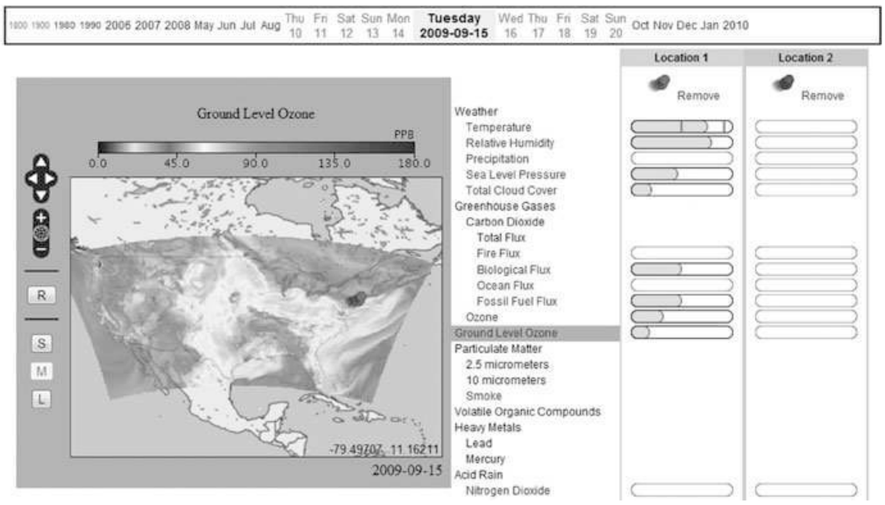


Fig. 10.1 A screen-shot of the Climate Data Explorer (CDX) application, which provides an integrated and interactive view of a large and diverse collection of datasets. CDX combines REST and modern dynamic client features using the RFDE framework

Various government and other organizations routinely publish data with direct relevance to climate. Examples of such organizations in the U.S. include the Environmental Protection Agency, the National Oceanic and Atmospheric Administration, and various state agencies such as the Maine Department of Environmental Protection. Data from these organizations differs in format and encoding, spatial and temporal coverage, measured or modeled attributes, and several other characteristics. As a result, it is difficult even for specialists to effectively use this data, even though most of it is publicly available on the Web. For example, a record of the global temperature and humidity fields for, say, December 31, 1984 is conceptually trivial to obtain based on datasets available on the Web. However, actually generating a suitable map-based representation of these fields is a difficult, laborious, and time-consuming (several hours) task for a specialist, and completely unworkable for a nonspecialist. In CDX, this representation may be generated in a matter of seconds using only a few mouse clicks and with no need for specialized knowledge.

Figure 10.1 depicts a screen-shot of the CDX Web application, illustrating its use for exploring climate data on a world map. For clients that support the required capability (mainly, modern JavaScript), the map uses common map features such as the ability to click and drag the map in order to pan around the globe, and balloon windows providing instantaneous feedback with more information on a clicked feature.

Some of the other components used by the CDX application include a *historical graph* and a *level indicator*. At the broadest level of client compatibility, these controls are both implemented using static images with hyperlinks to new windows containing additional or explanatory information. When clients support more advanced browser features, these components are rendered using SVG and support animation, panning, mouse-over tooltips, and other advanced usability features. When a new value is displayed in a level indicator (see Fig. 10.5 on page 48 for an example indicator), the horizontal bar is animated as filling from left to right, and the color changes as values transition from healthy to unhealthy ranges. The historical graph allows the user to pan the visible area of a very long time line. This is accomplished by clicking and dragging the display when supported, or by clicking on panning control buttons when the browser does not support client-side rendering of the data.

While the advanced interface features are important for enhanced usability and for designing a compelling and attractive application, their use may be counterproductive if it were to lock out some, or many, users with low-powered computers, older browsers (or sometimes very new ones), or some mobile browsers from being able to view the same information. Having the ability to easily support clients with a varying array of capabilities is one of the most important and challenging requirements of this application, and one that motivates much of the work described in the rest of this chapter.

Target Applications

We outline some characteristics of the applications that are best suited to the RFDE framework, using the CDX application of "Motivating Case Study: A Climate Data Explorer" as a typical and concrete example. The target Web applications for RFDE are essentially those for which the three requirements of, briefly, *portability*, *interactivity*, and *scalability* are of primary importance. These requirements are elaborated below.

The portability requirement refers to the ability to run on numerous and diverse computing environments, including various combinations of hardware (desktop computers, smart phones, kiosks, and more), operating systems, and Web browsers. For our CDX example, this requirement is crucial in ensuring that the benefits of exploring climate data are available to as many people as possible, including those using older hardware and software, and those with special accessibility needs. A similar comment also applies to, say, a Web store that would like to attract as large a customer base as possible.

The interactivity requirement refers to the need to have a strong visual impact and maintain user interest, based on a dynamic interface design that includes familiar modern Web widgets and provides instant feedback to user actions. Examples of these widgets include ones for browsing tiled maps, updating lists and selections based on user actions, and displaying pop-up windows with hints and error messages. Also included are widgets designed primarily to provide a visually pleasing experience, such as those for providing smooth transitions between images, and fade-in and -out of displayed items. While it may be tempting to write off the latter as frivolous decorations, their presence often makes a significant difference to the overall success and user acceptance of the application. For the CDX application, for instance, retaining user interest to encourage progressively more detailed exploration of the datasets and the underlying scientific and societal issues is greatly aided by such widgets.

The scalability requirement refers to the ability to easily increase the number of concurrent users supported by an application over several orders of magnitudes. For the CDX application, it is important that the implementation scale easily from hundreds to several tens of thousands of users as interest in the application grows and, further, that this scalability be achieved in a predictable manner by incorporating more hardware resources but without any significant qualitative change in the core design.

The portability and scalability requirements argue for the use of well documented and widely implemented Web standards. In particular, the REST approach is very natural and attractive design choice. The interactivity requirement argues for the use of modern Web widgets, tools, and JavaScript libraries that take advantage of recent developments in various parts of the client computing environment. Unfortunately, these two design choices are, without further work, largely incompatible. The core REST design and its typical implementations are based on the early interaction model between Web clients and servers, where most client actions generate a round-trip to the server, with concomitant implications for response times. Further, it is not immediately clear how one may apply the REST design to a Web application in which many actions, and state changes, occur through mechanisms such as Ajax (Asynchronous JavaScript and XML). This apparent incompatibility and its resolution are the core topics addressed by the RFDE framework, and this chapter.

While the RFDE approach itself is not dependent on any specific programming languages, scripting libraries, or client technologies, our implementation of the RFDE framework built to support the CDX application uses a number of specific languages that we will use in the examples throughout this chapter. Server-side code is written in the Java programming language, and client-side libraries and dynamically generated scripts are written in the JavaScript scripting language.

Widgets

The fundamental *resource* (in REST terminology) used by RFDE is the *widget*, which is a reusable user-interface element that allows one to view and manipulate application data. Common Web application widgets include form-entry fields, buttons, pull-down menus, checkboxes, radio buttons, and images. Widgets can also be built using other widgets, allowing for more complicated interface elements to be created quickly from the existing library, while also reducing proliferation of very similar code. By building a large collection of widgets, both general purpose and application specific, we can quickly create new Web applications that are portable, interactive, and scalable.

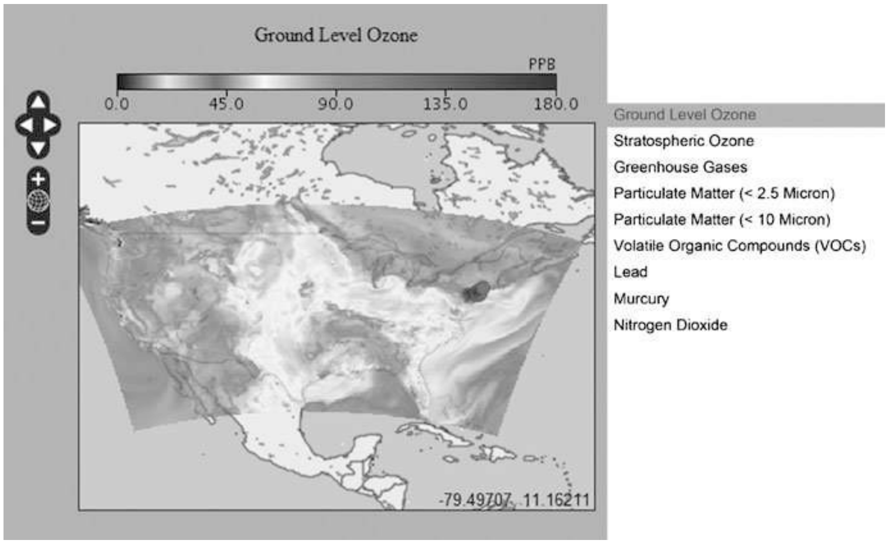


Fig. 10.2 A screen-shot depicting the use of the `mapview` application template in the CDX application

The CDX application (Motivating Case Study: A Climate Data Explorer) uses several application-specific widgets that allow the user to view and manipulate data from a multi-terabyte climate database. The screen-shot in Fig. 10.2 shows a simpler version of the climate-data browsing interface that consists of three primary widgets. The central widget is a *map widget* that supports the display of geographical distribution of the concentration of a climate parameter, such as the pollutant lead or stratospheric ozone. To the left of the map is a *navigation control widget*, consisting of several button widgets, that enables the user to pan and zoom the map. The third widget, displayed as the list of climate parameters to the right of the map, is a *selectable-list widget* that permits the selection of a parameter to display on the map.

A widget is implemented using one or more representations (e.g., a static image, DHTML, Flash, etc.) that correspond to the *capability set* (Client Capability Tiers) of the client. In the CDX application, the map widget is represented using the OpenLayers JavaScript library when the client supports it; otherwise, it is represented using a static image rendered on the server-side. Likewise, the map navigation buttons and the climate variable list items are represented using HTML anchor tags when JavaScript is not available, and as JavaScript supported clickable markup when it is.

Widgets can perform tasks through invokable methods and registered event handlers. The map widget is implemented using several methods such as `moveNorth`, `moveSouth`, `moveEast`, and `moveWest` which pan the map in the given direction; `center`, which centers the map on a given latitude and longitude; and methods to control the zoom level such as `zoomIn`, `zoomOut`, and `zoomWorld`. Each of these methods have dual implementations in the CDX library: one in Java that implements the method on the server, and one in JavaScript that can be invoked directly on the client when the widget is represented using the OpenLayers library (in general, each additional tier would require another implementation of the widget class). In addition to its methods, a widget can also identify a set of events that it generates. An event typically corresponds to a user action, such as changing the zoom level of the map widget (an `onZoom` event), and an application can specify what actions are performed when a given event occurs. Further details on methods and event handling, including examples, appear in "Event Handling".

The RFDE server publishes a common widget interface that can be used to obtain the value of a specific widget (that has a derived value) given a set of parameters. The value of the widget is represented using a language that is appropriate for programmatic use, such as such as XML or JavaScript Object Notation (JSON) (Crockford 2006). Later, we will discuss how this interface is used to implement much of the application dynamically, on the client side, when this feature is supported by the client environment.

Application Templates

An RFDE Web application is built using *application templates*, each of which is a composite resource (in REST terminology) that consists of collections of widgets that implement a common application usage pattern. In addition to its widgets, an application template also encodes the logic that controls the behavior of the widgets in the context of the template. A Web application contains only one instance of each application template, although a template may be replicated on multiple servers for load sharing.

An important property of RFDE templates, and one required by REST, is that the server side of an application does not save the state of a template for any of its clients. Instead, the client sends a request to the template that includes an encoding of its state, and the template returns a representation of the application at that state. For example, the CDX application uses a `mapview` template as suggested by Fig. 10.2. This template is initialized to a specific location, zoom level, and climate parameter; however, by manipulating the state value in the URI of the application, the client can change what information is displayed on the map.

The definition of the example `mapview` template is given in Listing 10.1. In lines 1–2, the template is created and assigned a CSS style sheet. The map widget and its corresponding navigation widget are created in lines 4–6. The navigation widget combines all of the map navigation buttons and automatically adds event handlers that invoke the corresponding methods of the map widget. Next, the list of parameters is created and populated with all of the possible variables that can be displayed on the map. In lines 14–15, an action is added to the list widget's `onChange` event handler that causes the `parameter` state variable of the map widget to be changed when the user selects a new value from the list. Finally, the widgets are added to the template (using a horizontal panel) and the template is initialized. This initialization routine involves the generation and caching of

```
template = new AppTemplate("CDX mapview Example", "mapview");
template.addStyleSheet("cdx");

MapWidget map =
   new MapWidget(40.7166, -74.0067, 1, 400, 300, "o3");
MapNavigator nav = new MapNavigator(map);

List plist = new List();
plist.addItem("Ground Level Ozone", "o3");
plist.addItem("Stratospheric Ozone", "o3strat");
// ... additional values omitted ...
plist.addItem("Nitrogen Dioxide", "no2");

plist.onChange().addAction(
   new StateChangeAction(
      map, "parameter", plist, "selectedValue"));

template.addWidget(new HorizontalPanel(nav, map, plist));
template.init();
```

Listing 10.1 The definition of the `mapview` application template

static markup that will be used in every document generated by the template; the creation of an explicit representation of the default state of the template, based on the parameters specified in the template definition (Representation of Application State); and the use of a widget dependency graph to create a valid ordering for instantiation in client-side code.

When a new template request is made, the server program that hosts the application is responsible for translating the encoded application state into a state object, "executing" the template, and returning the resulting document. Executing a template requires generating the markup language for each of the widgets based on the current state of the application, as well as creating initialization parameters for client-side versions of the widget implementation classes. The resulting document contains static references to external resources used by the document (such as style sheets), references to the RFDE libraries that implement the client-side versions of the widget classes used by the template, the generated upgrade parameters and event handlers, and finally, the markup the implements the page and its widgets (example markup for an *image push button* widget is given on page 247).

Client Capability Tiers

The RFDE framework supports the development and deployment of Web applications that support, concurrently and interchangeably, client environments with diverse and changing capabilities. For instance, one user may run the application on

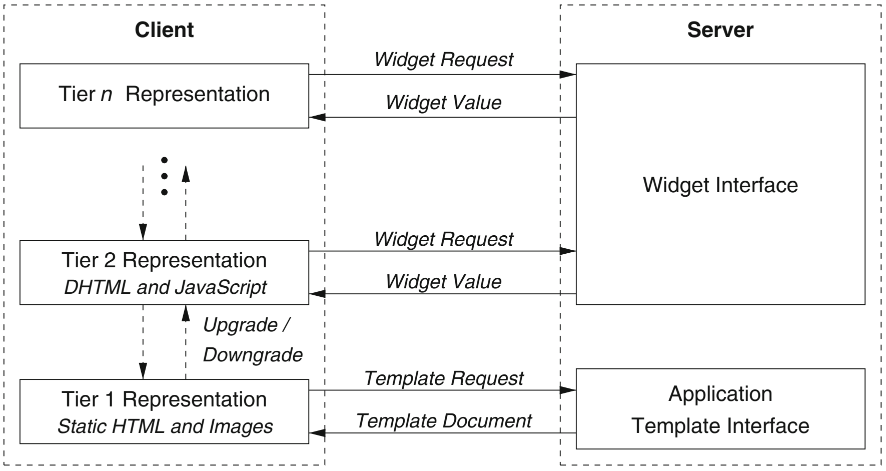


Fig. 10.3 Client capability tiers in RFDE

a desktop computer running Windows XP and Internet Explorer 8 while others (or the same user) access it using, variously, a smart phone running Symbian and Opera, a kiosk running GNU/Linux and a customized version of Firefox, or a computer with software that is several years behind the current versions.

It would be foolhardy to attempt to explicitly address every possible combination of the components of a client environment: hardware, operating system, Web browser, and so on. Instead, RFDE models the features and abilities of the computing environment on the client side using *client capability tiers*. These tiers classify client environments by specifying the properties required for tier membership. RFDE includes a default definition of these tiers, but application programmers may easily modify both the number of tiers and the individual tier definitions, and such modification is expected and encouraged. The lowest tier (Tier 1) is designed to be as inclusive as possible, and thus specifies the bare minimum for what is needed for the application to function. A guideline for Tier 1 is to include only those requirements without which there is no reasonable way to accomplish the key tasks of the application. As suggested by Fig. 10.3, each higher tier adds increasingly demanding requirements for the client environment. When a client interacts with an RFDE application, the framework automatically uses the highest (most capable) tier that the client's environment supports. This default behavior may be changed, and the tier may be explicitly set to a desired one by using *tier selection widgets* which are typically used during testing.

Tier 1 clients that support only the minimum requirements are able to use a fully-functional version of the Web application, although some of the visual and usability enhancements afforded by more capable environments may be missing. As a simple example, a client without scripting support may not provide immediate feedback on potentially incorrect data. However, not only are the functions implemented by the form (perhaps a purchase) fully supported, but also the feedback on incorrect data is provided, albeit with a slightly longer response time due to a server round-trip and page refresh. If the client environment supports additional capabilities, the application widgets will be automatically *upgraded* to versions that use these capabilities to improve the speed, responsiveness, usability, or appearance of the application. A special JavaScript class in the client-side RFDE library, called the *widget manager*, is responsible for the instantiation and automatic upgrade of all of the widgets in an application document based on the identified tier level of the client environment.

For the CDX application, consider the `mapview` template of Fig. 10.2. In Tier 1, the user is able to pan and zoom the map but must do so using the navigation buttons on the left. A more direct manipulation of the map by clicking and dragging on the map itself is not supported because the client environment capabilities (JavaScript, etc.) that are needed to implement such manipulation are not part of Tier 1. Map manipulations, and most other actions, in this tier also require full page refreshes and a new rendering of the visible area of the map, with the associated, typically noticeable, delays. In Tier 2, the map is more interactive. In addition to the direct manipulation using dragging, it also permits zooming in and out using scroll wheels and similar input modes. Further, map features are associated with pop-up balloon windows with hints or other brief messages. Map tiles and other images are loaded asynchronously and partial updates of the displayed Web page are accomplished by manipulating the DOM tree; these enhancements avoid full page refreshes in most cases and so greatly improve responsiveness.

Figure 10.3 depicts this tiered approach of a RFDE Web application. At the lowest level of the figure, a client communicates with the application server to request an updated view of the application. At this level, the document returned contains the entire application template, including all the widgets in the template. The state of the application is explicitly encoded in the URI that the client sends, and the application view that is returned is represented using a markup language such as HTML. The hyperlinks in the document contain URIs that encode new states for the application, so that when the user clicks on a link, the net effect is that the state of the application is updated and the new view of the data is returned.

Embedded in a Tier 1 client document is a small script that checks client capabilities when the document is loaded. If the client does not support the scripting code, it will simply be ignored and the client will remain at this tier for the duration of the exchange. If the capability check determines that the browser supports a higher tier, the client-side widget manager will automatically upgrade all the widgets on the page to their higher-tier representations. For example, Tier 1 may represent the application using HTML and static images, Tier 2 may add JavaScript and client-rendered images, and a third tier may use Adobe Flash or advanced SVG graphics to render the application. If the client supports JavaScript, but not Flash or advanced SVG, the client code will upgrade the widgets to their Tier 2 versions and future interactions with the server will take place at the RFDE widget interface.

In addition to the server-side Tier 1 widget library, an RFDE server supports an arbitrary number of additional levels of higher-capability, client-side widget libraries. At these higher tiers, the client requests the value of individual widgets, instead of entire application templates, through the common widget interface. This design allows the client to use asynchronous transactions to replace the value for individual widgets in a template, improving the application's responsiveness. The upper-tier widget libraries use representations for widget values that are more appropriate than HTML, such as JSON, allowing for any type of client technology (such as HTML, DHTML, SVG, Flash, etc.) to be used to render the widget.

The initial framework developed for the CDX application consists of two tiers of client capability. However, additional tiers are likely to be added based on the expected mix of client categories and an important aspect of RFDE is that such additions can be made easily, without affecting existing code and application functionality. In the lowest tier, the widgets are represented using HTML 3.2, static images, image maps, and hyperlinks. The application also uses cascading style sheets to control the look and feel of the page. These style sheets are ignored by browsers that do not support them. Images, such as the tiles in the visible area of the map, are rendered by the server and sent to the client in a widely supported format such as JPEG, GIF, or PNG. In this level, each user interaction with the application (informally, each click) requires a complete page refresh. For example, a single-button widget, such as the zoom-in button in the map navigation control widget, is represented using the following HTML:

```
<a id="ImagePushButton5"
   class="ImagePushButton ImagePushButton-t1"
   href="/cdx/1.0/mapview?state=&e5=zoomIn">
  <img src="/images/map/zoom-in.jpg"
       alt="Zoom In"
       border="0" />
</a>
```

The widget is represented as a simple hyperlinked image in this tier. When the user clicks on the image, indicating a zoom-in event, the state of the application is updated (in a REST-compatible manner) following a round-trip interaction with the application server and subsequent page refresh at the client. Event handling is discussed further in "Event Handling".

If the client supports JavaScript, DHTML, Ajax, and SVG, it is automatically promoted to second tier functionality when the application is loaded. In this tier, each upgradeable widget is replaced with its JavaScript and DHTML implementation. After such an upgrade, client interactions no longer require a full page refresh. Widgets change their displayed forms by using client technologies, such as JavaScript and DHTML. For example, the upgrade dynamically replaces the earlier static-HTML representation of the zoom-in button with its second tier equivalent:

```
<img id="ImagePushButton5"
     alt="Zoom In"
     src="/images/map/zoom-in.jpg"
     class="ImagePushButton ImagePushButton-t2">
```

Unlike the earlier representation, there is no longer a static hyperlink and the widget identifier now appears in the image tag. The `ImagePushButton-t1` CSS class has been replaced with the `ImagePushButton-t2` class, allowing for independent styling of the two tiers. When upgraded, a JavaScript-class implementation of the widget is instantiated and the class registers any required event handlers (such as `onClick` for this button) with the browser. If a widget has a derived value, a value that is determined by its parameters that is also dependent on other information, such as a database, the widget will update its value using an asynchronous callback to the RFDE widget interface. These changes allow a control to remain dynamic without requiring the full-page refresh caused by the hyperlink-based implementation. The two representations of the zoom-in button are visually and functionally nearly identical; however, in Tier 1 pressing the zoom in button requires a complete page refresh to perform the operation, while in Tier 2, the event is handled completely in the browser without requiring a page refresh.

Representation of Application State

Following REST conventions, the current state of an RFDE Web application is explicitly encoded in the application's URIs. The advantages of this design are similar to those of other REST-based ones: By using a completely stateless protocol, multiple servers can implement the application, client requests can be handled by any available server, and the application can be scaled by increasing the number of available servers in a load-sharing environment. This design also allows the use of caching strategies to optimize common requests, such as the most recent map images for frequently queried areas of the United States. Finally, by explicitly representing the state of the application in the URI, users of the application can bookmark and revisit a particular view of the application, or share their experiences with others, in a robust and standard manner.

Our implementation of RFDE identifies state variables using a positional scheme in order to reduce the total size of the state encoding (compared to an alternative named-variable scheme, as used for HTML query strings). To further reduce the size of the state string, values that have not changed from their template-specific default values are omitted from the encoding. The state of each widget (the collection of its parameters) is represented as a string composed of the widget identifier followed by a colon delimited list of the state values. The state of the entire application template consists of an asterisk-delimited list of widget states. In order to support long-term bookmark compatibility as an application and its widgets evolve over time, each application template URI includes the application version. When an application receives a request with an old version number, it should attempt to construct an equivalent URI compatible with the latest version and redirect the client (using an HTTP 301 Moved Permanent redirection).

Figure 10.4 shows an example hierarchical state representation for the `mapview` application template from Fig. 10.2. This application template consists of three widgets; however, the map navigation widget does not have any internal state and is omitted from the state representation. The map widget has the `Map1` identifier and the selectable list of climate parameters is given the identifier `List1`. The default

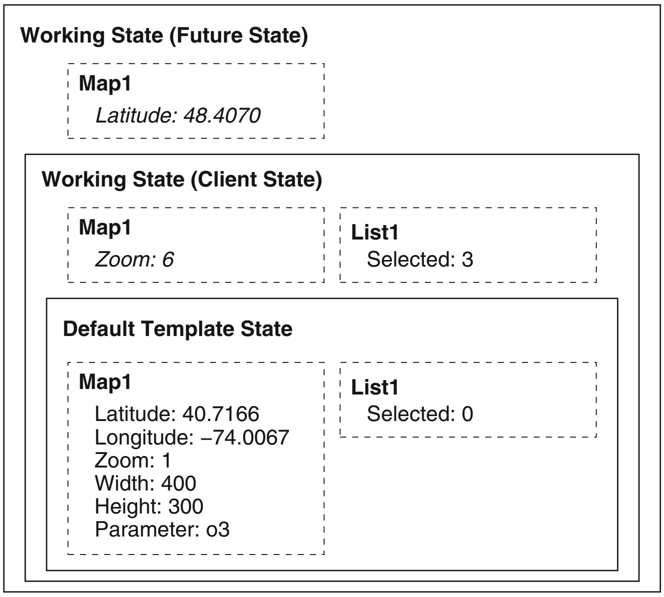


Fig. 10.4 An example of the representation of an application template's state. The innermost state represents the application's default state while the outer states are specific to a client request

application state is shown in the innermost layer of the diagram, which contains values for all of the properties for the two widgets. The order of the properties in the diagram corresponds to the order of the values in the state value string, so latitude is the first, longitude is the second, and so on. When the client does not specify a value for the application state, the default state is used (1.0 identifies the version of the web application):

```
/cdx/1.0/mapview
```

A client may also use the following complete state representation, even when the application is at its default state.

```
/cdx/1.0/mapview?state=Map1:40.7166:-74.0067:1:400:300:o3*List1:0
```

A *working state* is a representation of state that keeps track of changes from another state (typically the default state, but working states may also be nested). When the client sends a request for an application template, the server builds the working state for the request which is then used to generate the document that is returned. If a state variable is not included in a working state, the default value for the variable is used. The middle layer in Fig. 10.4 represents the current client state, in which the values of two state variables have been changed from their defaults. This state is created in response to a client request with the following application URI:

```
/cdx/1.0/mapview?state=Map1:::6*List1:3
```

The only changed property of the map widget is the zoom level, which is the third property of the widget. The colons corresponding to the first two properties of the map widget must be included in the encoding to ensure proper positional representation; however, additional colons at the end of an encoding may be dropped. In this example, the colons corresponding to the last three properties (width, height, and parameter) are dropped because these properties retain their default values.

The outermost working state in Fig. 10.4 represents a potential future state that may be used to generate proper URIs for inclusion in the current application hypertext. In this example, this future state represents the state of the application if the user were to pan the map to the north, and this state could be encoded in the hyperlink URI for the corresponding map control button:

```
/cdx/1.0/mapview?state=Map1:48.4070::6*List1:3
```

The default state representation for an application template is a constant value that is only initialized once, when the template is created, and then shared among all client requests. When the server receives a new request, it only has to instantiate a more light-weight working state to represent the changes from the default state. When a working state is created, the RFDE application server automatically performs type and sanity checking of the state values based on constraints that can be specified when a widget registers a new state variable in the default state.

While operating at the lowest tier level, the client manages the application state implicitly, using state-encoded URIs in hyperlinks and HTML forms. When a client is upgraded to a higher tier level; however, the client becomes more actively responsible for keeping track of the state of the application. Many of the actions that are performed by an event handler are simple to complete on the client, such as changing the CSS classes used by the selectable-list widget in order to highlight a newly selected value, and requiring a round-trip exchange with the server in order to perform this task would be an unnecessary cause of latency that would affect the perceived responsiveness of the application.

The widget manager is responsible for keeping track of the application state on the client. While the server needs to explicitly model the default state and any changes to the default state made by each of the clients, the client only needs to keep track of the current state of the application. When the application state is changed, the widget manager requests any updated widget values from the server (if necessary) and updates the current application URI to allow the user to bookmark any particular view of the application.

Event Handling

We use the term *events* to refer to the interactions of a Web application user with the user interface. Examples of events include clicking on buttons, selecting items in drop-down menus, and panning a map. Each application widget recognizes the events that relate to it. For example, a map may have an onMove event which corresponds to a user request for panning to a new location and an onZoom event which corresponds to a user request for changing the zoom level. Each event may be associated with a set of actions that are performed whenever the event occurs, and these actions may in turn affect other widgets in the application. A simple example in the CDX mapview template is that changing the selected climate parameter in the list widget also changes the parameter that is displayed in the map.

After creating the widgets in an application template, the programmer specifies the application behavior by associating actions with widget events. Actions can affect an application in various ways, such as changing a state variable, invoking a widget's method, or even firing another event, which may in turn trigger additional actions, recursively. For example, the zoom-out button in the navigation widget of the CDX application is assigned an action that invokes the map's zoomOut method when the user clicks on the button:

```
zoomOutButton.onClick().addAction(
   new InvokeMethodAction(map, "zoomOut")
);
```

The manner in which this event handler is executed depends on the client capability tier (Client Capability Tiers) that is active at the time of the event.

In Tier 1, an event is initiated by including an event identifier in a request query string. Events have an optional argument which is used to specify event parameters. For example, an event caused by the user selecting a different climate variable would be parameterized with the index of the new selection. When there is no actual parameter, the value 1 is used to indicate that the event was activated. In the CDX application, the zoom-out button's onClick event is assigned the identifier e3 and the hyperlink has the following URI:

```
/cdx/1.0/mapview?state=List1:2&e3=1
```

This URI indicates that the only change from the default state of the mapview template is that the third climate parameter is selected in List1 (using zero-based indexing) and that the zoom-out button has been pushed.

When the server receives a request that includes an event identifier, it immediately triggers the associated event handler. In this case, the only associated action is to execute the zoomOut method of the map widget, as specified by the following server-side code fragment:

```
public void zoomOut(WorkingState state, String param) {
   int zoom = state.getIntegerValue(getId(), "zoom");

   // Update the zoom state variable
   state.setStateVariable(getId(), "zoom",
          (int) Math.max(0, zoom - 1));

   onZoom().fireEvent(state, "out");
}
```

Since templates are stateless, the current application state is passed as the argument state to the zoomOut method. The method determines the current value of the zoom, updates it, and modifies the working state. Finally, the method triggers the map widget's `onZoom` event which, by similar mechanisms, triggers the appropriate event-handling method for the map widget, which will cause any actions identified as side-effects to changing the zoom level to be also be executed (there are none in the `mapview` example template).

Once the server has completed executing all of the event handlers, the client is immediately redirected, using an HTTP 303 See Other redirect, to a URI that fully encodes the new application state based on the updated value of the working state. Thus, the non-transient URIs at the client never include pending events. As a result of the zoom-out widget's `onClick` event, the client is redirected to the following URI with a modified zoom value:

```
/cdx/1.0/mapview?state=Map1:::0&List1:1
```

At higher tier levels, more of the event handling is managed on the client side in order to increase the responsiveness of the application and to reduce the number of complete page refreshes. Tier 2 event handling in RFDE is performed by the JavaScript implementations of the widgets. When widgets are initialized, they are given JavaScript versions of event handlers. The zoom-out button is instantiated with the following event handler, which is automatically generated from the Java version of the event handler (the `$I` function returns the instance of the identified widget):

```
onClick: function(param) {
    $I('Map1').zoomOut(param);
}
```

The client-side JavaScript version of the map widget has the following implementation of the `zoomOut` method:

```
zoomOut: function(param) {

    // Update the zoom state variable
    this.state.zoom = max(0, this.state.zoom - 1);
    this.state.update()

    // Zoom out the JavaScript map
    this.map.setZoom(this.state.zoom);

    this.onZoom("out");
}
```

This client-side implementation of `zoomOut` is nearly identical to the earlier server-side implementation, but there are two notable differences: First, the widget manages its own state directly rather than requiring the state as an additional argument. Second, and more important, the client-side version of the method actually causes the map to zoom out as a direct side-effect. Recall that the server-side version only modifies the representation of the application state.

When a widget needs to update its value due to an event that is handled on the client side, it requests the value from the RFDE server's widget interface, based on its updated parameters. For example, one of the widgets in the CDX application is a *level indicator*, a widget that graphically presents the value and health implications of a specified climate parameter, such as a pollutant, at a location and time which

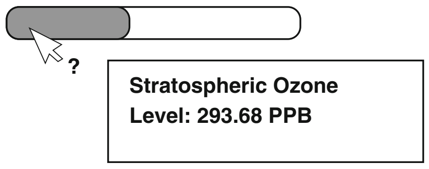


Fig. 10.5 An example of the level indicator widget which, when upgraded, uses asynchronous calls to the server to modify its value as a user changes the selected location or date being displayed

are specified using other widgets. Figure 10.5 depicts an example level indicator for stratospheric ozone (the ozone layer). The horizontal bar in the figure is filled to indicate the comparative value of the underlying parameter. The bar's color is mapped to health standards, with green denoting a healthy level, for instance. In our Tier 1 implementation of the level indicator widget, it is rendered as a static image generated on the server side. When the client is upgraded to Tier 2, the indicator is rendered on the client side and gains niceties such as animated filling of the bar and a textual description of the level that appears as a balloon activated by a pointer-hovering event.

When the user changes the selected date or location, the level indicator must be updated to display the parameter value at the date or location. The widget sends an asynchronous request for the new value to the widget library. This REST-based interface can supply the value of a widget based on the widget's parameters, in a representation that is more appropriate for programmatic manipulation. For example, the following URI requests an updated value for a level indicator widget:

```
/w/1.0/LevelIndicator?state=o3strat:48.41:-74.01:2010-09-23
```

The server responds with a representation of that value, in this case encoded using JSON:

```
{
  "widget"     : "LevelIndicator",
  "version"    : 1.0,
  "state"      : { "parameter" : "o3strat" ,
                   "latitude"  : 48.41,
                   "longitude" : -74.01,
                   "date"      : "2010-09-23" },
  "uri"        : "/w/1.0/LevelIndicator?state=
                    o3strat:48.41:-74.01:2010-09-23",
  "param_name" : "Stratospheric Ozone",
  "param_alt"  : "The Ozone Layer",
  "units"      : "PPB",
  "level"      : 293.68,
  "US_limit"   : undefined,
  "EU_limit"   : undefined,
  "health_idx" : 0
}
```

On receiving this response from the server, the client-side widget code changes its internally stored value and re-animates the filling of the display bar.

A Sample User Session

We now illustrate some of the interactions outlined in earlier sections in the context of a simple session of user interactions with the CDX application of "Motivating Case Study: A Climate Data Explorer." First, the client loads the CDX portal, which is a directory for a number of CDX application templates, by sending a request to the server. Next, the user selects a link to the `mapview` application template. Finally, the user performs two events on this application page: (1) changing the selected parameter in the list to lead and (2) activating the zoom out control for the map.

Figure 10.6 illustrates this sequence of events for both Tier 1 and Tier 2 compatible clients. For simplicity, this figure focuses entirely on the interactions that represent the main application logic flow; not shown are the additional requests for document resources, such as embedded images, made by the client. The Tier 1 interactions are shown in Fig. 10.6 (a). The first two user requests (for the CDX portal and the `mapview` template) are made as standard HTTP GET requests; each results in the server generating and returning a complete Tier 1 document. The event

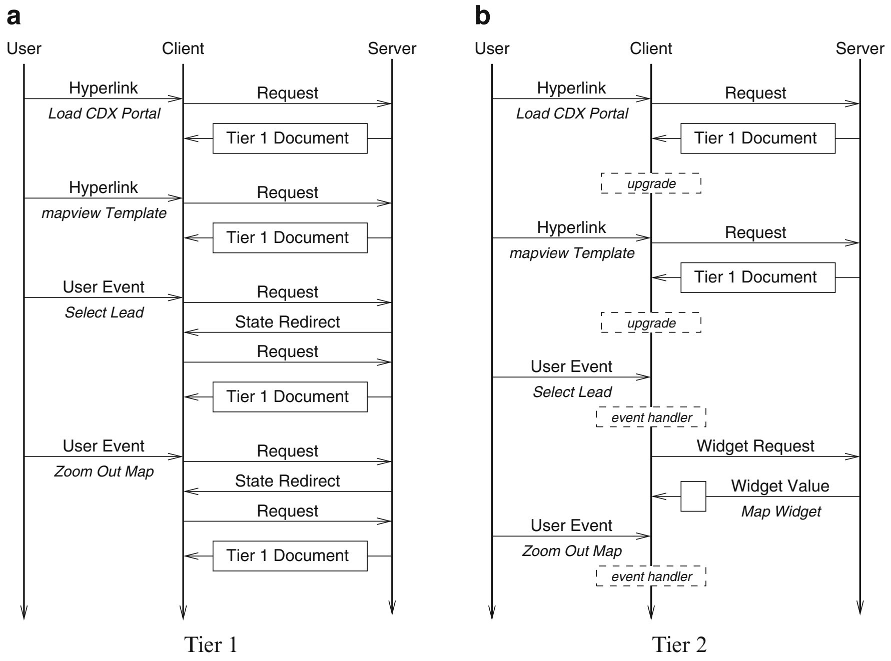


Fig. 10.6 An simple CDX session. At Tier 1, each request and event requires a full-page refresh. Event handlers on the server compute the modified state representation and redirect the client to the new URI. At Tier 2, event handling is performed on the client and only the values of individual widgets are requested from the server and updated on the client side. Some events may be handled completely on the client and do not require a request to the server (such as the `onZoom` event corresponding to the user zooming out the map in this example)

requests (selecting lead and zooming out) require server-side event handling. For both of these requests, the server receives the request from the client which includes the event identifier, executes the event handler which computes the new application state, and then redirects the client to the new application URI, which encodes the new state.

The corresponding sequence of events for a Tier 2 client is shown in Fig. 10.6 (b). For complete template requests, the Tier 2 interactions are handled exactly as in Tier 1; however, when the client loads a Tier 1 document, the embedded script upgrades the document to its Tier 2 equivalent. In Tier 2, event handling is performed on the client side, rather than requiring a complete page refresh, reducing latency and allowing the application a much greater level of responsiveness. When the user changes the parameter to lead, the event handler for the selectable list widget's `onChange` event (in client-side code) signals the widget manager to asynchronously request a new value for the map widget. The response from the widget interface, which is significantly smaller than a complete Tier 1 document, includes details that the map widget requires to properly render the new map and to request a new set of map tiles. Some events, such as when the user zooms the map out, can be handled completely on the client, and do not even require a request for an updated widget value. The underlying map tiles are requested from the server as usual, although they may also be cached on the client side by the usual browser mechanisms.

Related Work and Discussion

The RFDE framework described in this chapter is an advanced and REST-based *progressive enhancement* strategy (Wells and Draganova 2007; Parker et al. 2010) for Web development. This strategy uses, at the core, basic markup that is supported by the capabilities of the most primitive expected client. Advanced features and layout implemented through external links to JavaScript and Cascading Style Sheets. Progressive enhancement is based on the separation of document structure from the layout styling, and all presentation tasks are handled by style sheets. In contrast to strategies based on *graceful degradation* (Randell et al. 1978), which degrade to a more basic implementation when the client does not support the full implementation, the progressive enhancement strategy ensures that any client always obtains the full content and at least a minimal set of functionality and styling. This strategy is especially important for ease of indexing by search engines, and for users of assistive technologies which typically require that the basic content is always available and not hindered by dynamic content delivery.

With the development of mobile Internet devices such as smart phones, eReaders, and tablets, there has been a large amount of work on multi-device user interfaces (e.g., Grundy and Yang 2003; de Oliveira and da Rocha 2005) that allow an application to use the native features of the host device. Many of these approaches have adopted a device-independent user-interface specification language such as

UIML (Edwards et al. 2000; Ali and Abrams 2001), and use an application-independent user interface library to realize the application on the host device.

Nokia has described a Remote MVC (model-view-controller) application controller (Stirbu 2010) that models user interfaces as REST resources and that uses an event-based system to keep the client and service synchronized. When this framework is initialized, applications can discover and acquire platform-specific representations of the user interface elements that use a device's native functionality and match the look-and-feel of the device.

A significant advantage of the RFDE framework of this chapter is that it allows the development of portable, interactive, and scalable applications. Rather than attempt to support the myriad of client devices natively, RFDE models common client features in tiers of capability. These tiers allow the development of specific representations for interface elements based on a small number of tiers that framework implementations support, rather than developing a representation for each possible device.

While our description in this chapter uses several specific technologies such as JavaScript and Java, the RFDE design does not depend in any significant manner on these technologies, and others may be substituted where appropriate. The framework also does not impose an architectural style (such as MVC) on an application, and the programmer may choose the one best suited to a task.

Application programmers who use RFDE can develop an application by adding widgets to an application template and then specifying actions that should occur as a result of their related events. This development approach is similar to application development for desktop applications and Web development frameworks that are modeled on desktop application development, such as the Google Web Toolkit (McFall and Cusack 2009). In this approach, the application programmer may treat widgets as abstract entities without immediate concern to their implementation on various client environments.

A major goal of this approach is to develop user interfaces that can become more responsive and intuitive according to the capabilities of the client environment, while affording all of the functionality of the application, even at the lowest level of capabilities. Here, the functionality of the application refers to what can be done with the application and not necessarily how it is performed. In the CDX application, for example, the user must use the buttons in order to navigate the map at Tier 1, but at Tier 2, the map becomes responsive to mouse control. Tier 1 users can still fully navigate the map, even though they need to do so via a slightly more primitive interface and changes require a full request/response cycle with the server. Tier 2 map users can still use the familiar button interface to navigate the map and both visual representations (assuming that basic CSS is supported by the Tier 1 clients) of the maps are identical at both tiers.

While the RFDE approach removes, or at least significantly reduces, the need for a Web application developer to produce device or platform specific interface elements, it does not preclude the development of native interface implementations. In fact, the REST uniform interface constraint supports and facilitates the development of these native interfaces. Devices can use their own implementations of the interface widgets and populate any derived values via requests through the widget interface, in the same way that the CDX Tier 2 clients do. Optimized clients for the Climate Data Explorer are currently under development for several popular mobile device platforms.

The current version of the RFDE server implementation does not include a uniform and generic method for describing the logic of an application template (e.g., the widgets, layout, constraints, and event handlers that comprise the template); however, the development of such a language is an important next step in our research. This language would allow native implementations to utilize the same application templates that the Web applications use and reduce the development costs associated with adapting new application templates and updating existing templates as they are improved.

One potential drawback to the RFDE approach is that each widget needs to be implemented multiple times. For example, in order to create a widget for CDX, a Java class that implements the Tier 1 widget needs to be written, a JavaScript analog needs to be written for the Tier 2 client-side implementation, and (for some of the widgets) the RFDE widget interface needs to be updated to generate a JSON representation for the value of the widget. On the other hand, once the underlying widgets have been implemented, an application can be developed that automatically supports the various levels of capabilities of its clients – the alternative would be to develop multiple versions of the same application. One solution to the problem of handling the dual implementation of the widget library, and one that we plan on investigating for the next version of the framework, is the development of a language or library that can be used to write widgets that will automatically compile both the client-side JavaScript libraries as well as the server-side implementation from a single source.

Acknowledgements This work was supported in part by the U.S. National Science Foundation grant EAR-1027960 and the University of Maine.

References

Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs. “big” web services: making the right architectural decision. In *WWW ’08: Proceeding of the 17th International Conference on World Wide Web*, pages 805–814, ACM, New York, NY, USA, 2008.

Dean Jackson and Craig Northway. Scalable vector graphics (SVG) full 1.2 specification. WD not longer in development, W3C, April 2005. http://www.w3.org/TR/2005/WD-SVG12-20050413/.

Douglas Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational), 2006.

Jesse James Garrett. Ajax: A new approach to web applications. 2005.

John Grundy and Biao Yang. An environment for developing adaptive, multi-device user interfaces. In *AUIC ’03: Proceedings of the Fourth Australasian user Interface Conference on User Interfaces 2003*, pages 47–56, Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 2003.

John Wells and Chrisina Draganova. Progressive enhancement in the real world. In *HT '07: Proceedings of the Eighteenth Conference on Hypertext and Hypermedia*, pages 55–56, ACM, New York, NY, USA, 2007.

Mir Farooq Ali and Marc Abrams. Simplifying construction of multi-platform user interfaces using UIML. In *European Conference UIML*, 2001.

B. Randell, P. Lee, and P. C. Treleaven. Reliability issues in computing system design. *ACM Comput. Surveys (CSUR)*, 10(2): 123–165, 1978.

Rodrigo de Oliveira and Heloísa Vieira da Rocha. Towards an approach for multi-device interface design. In *WebMedia '05: Proceedings of the 11th Brazilian Symposium on Multimedia and the Web*, pages 1–3, ACM, New York, NY, USA, 2005.

Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2(2):115–150, 2002.

Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

Ryan McFall and Charles Cusack. Developing interactive web applications with the google web toolkit. *J. Comput. Small Coll.*, 25(1): 30–31, 2009.

Stephen Edwards, Manuel A. Prez-quiones, Mary Beth Rosson, Robert C. Williges, Constantinos Phanouriou, and Constantinos Phanouriou. UIML: A device-independent user interface markup language. Technical report, 2000.

Todd Parker, Scott Jehl, Maggie Costello Wachs, and Patty Toland. *Designing with Progressive Enhancement: Building the Web that Works for Everyone*. New Riders Publishing, Thousand Oaks, CA, USA, 2010.

Vlad Stirbu. A restful architecture for adaptive and multi-device application sharing. In *WS-REST '10: Proceedings of the First International Workshop on RESTful Design*, pages 62–66, ACM, New York, NY, USA, 2010.

Chapter 11
From Requirements to a RESTful Web Service: Engineering Content Oriented Web Services with REST

Petri Selonen

Abstract This chapter presents an approach for proceeding from a set of requirements to an implemented RESTful Web service for content oriented systems. The requirements are captured into a simple domain model and then refined into a resource model. The resource model re-organizes the domain concepts into addressable entities: resources and interconnecting links, hypermedia representations, URIs and default HTTP operations and status codes. The approach has emerged from the experiences gained during developing RESTful Web services at Nokia Research Center.

Introduction

REST and Resource Oriented Architecture (ROA) (Richardson and Ruby 2007) are particularly well suited for content oriented Web services whose core value are in storing, retrieving and managing interlinked content through a uniform interface. While REST has gained significant popularity as the architecture for Web services, there is a notable lack of methods and modeling notations for developing RESTful services from requirements. There is a need for communicating the requirements and design intent to the different stakeholders, as well as to map the requirements and new features to the existing implementation in a way that preserves consistency of the API and supports service evolution over time.

Probably the best known formulation of how to design RESTful Web services has been presented by Richardson and Ruby which involves finding resources and their relationships, selecting uniform operations and respective response codes for each resource, and defining their data formats. The formulation is too abstract to be followed as a method and does not facilitate communication between service

P. Selonen (✉)
Nokia Research Center, Visiokatu 1, Tampere 33720, Finland
e-mail: petri.selonen@nokia.com

architects and other stakeholders. Even with a priori content oriented services, it is often a non-trivial exercise to refine a functional specification to resource-oriented, descriptive state information content. According to our experiences, many developers find it hard to make a paradigm shift from object oriented design that emphasizes hiding state-related data behind task-specific operations of an instructive interface. Such API centric thinking has a tendency to make resulting Web services look more like a collection of unrelated APIs instead of a set of interlinked content accessible using a uniform interface.

This chapter presents a systematic but light-weight approach for proceeding from a set of requirements to an implemented RESTful Web service and integrating new requirements with an existing system for content oriented services. The requirements are first collected into a domain model – expressed with UML class diagrams – which is essentially a structural model describing the domain concepts, their properties and relationships, and annotated with information about required searching, filtering and other retrieval functionality, and constraints. The domain model is then gradually refined into a resource model that is used to derive the resources and interconnecting links, representations, assigned URIs, and default HTTP operations and response codes of the Web service. The concepts of a resource model can be further mapped to implementation level concepts in service specifications, database schema and source code.

The approach has emerged during the development of a RESTful Web service for building Mixed Reality services at Nokia Research Center and reported in Selonen et al. (2010a,b). The gained experiences suggest it supports architects' communication with the service clients, and perhaps more importantly, with software engineers not familiar with REST and ROA. The presented approach helps bringing in new features to the system in a non-intrusive way in a rapid pace while supporting traceability of requirements and actual implementation.

The chapter is organized as follows. In "Overview of the RESTifying Approach," we introduce the "RESTifying approach" with an overview of the process, domain models, resource models and discussion on how to refine the former into the latter. "Example Web Service: Social Points of Interest" gives a step-by-step example on how to apply the approach for a simple Web service starting with requirements and domain model, and refining the domain model into resource models and respective components of a RESTful Web service. "From Resource Model to Implementation" discusses how to implement the service defined by the resource model. Finally, "Concluding Remarks" gives concluding remarks.

Overview of the RESTifying Approach

While representing arbitrary functionality as uniform resource manipulations is hard, content oriented systems already exhibit resource oriented characteristics with some additional filtering and querying capabilities. With this notion, we claim that for such systems, a domain model can essentially capture most of the system

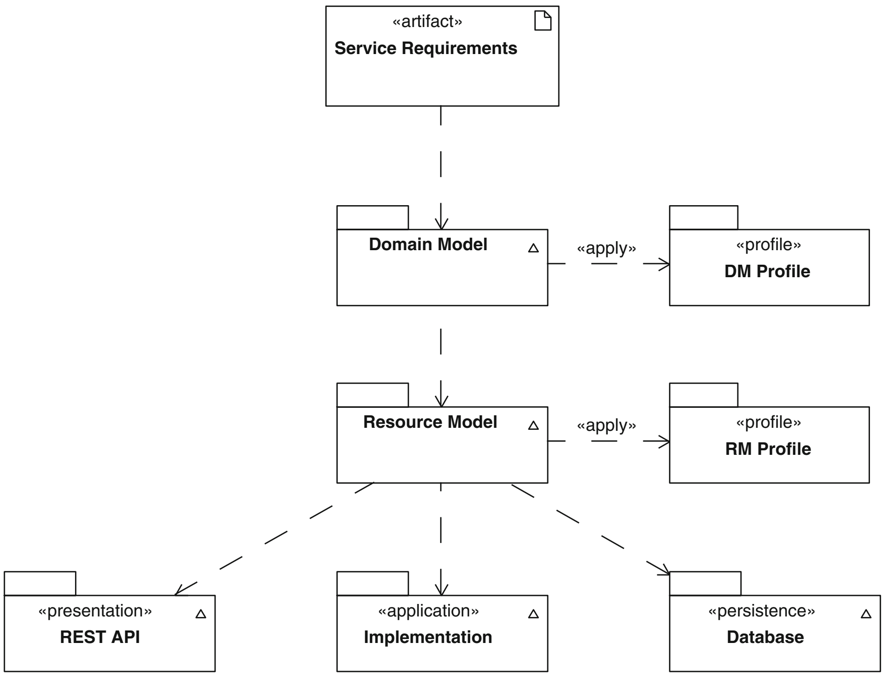


Fig. 11.1 The artifacts produced by the approach

requirements. In order to refine a domain model to a resource model, we identify a minimal set of modeling concepts that can then be mapped to implementation level concepts and hypermedia content offered through a RESTful interface.

Figure 11.1 shows the artifacts produced by the approach. The service requirements are captured into a domain model, expressed as UML class diagrams. The domain model is then refined into a resource model. A resource model organizes the concepts of the domain model to addressable entities that can be mapped to elements of a RESTful Web service interface, service implementation and database schema. We use UML profiles to constrain the elements in the domain model and information model.

Domain Model

The domain model describes concepts of a system, the attributes and attribute types related to the concepts, sub-concepts of the concepts and filtering criteria related to the concepts. As it represents the key concepts of the system and their attributes using the vocabulary of the problem domain, it can be used to communicate the system requirements among the system stakeholders, and as a starting point for software development. It is in principle a subset of a vanilla class diagram with a few additional annotations.

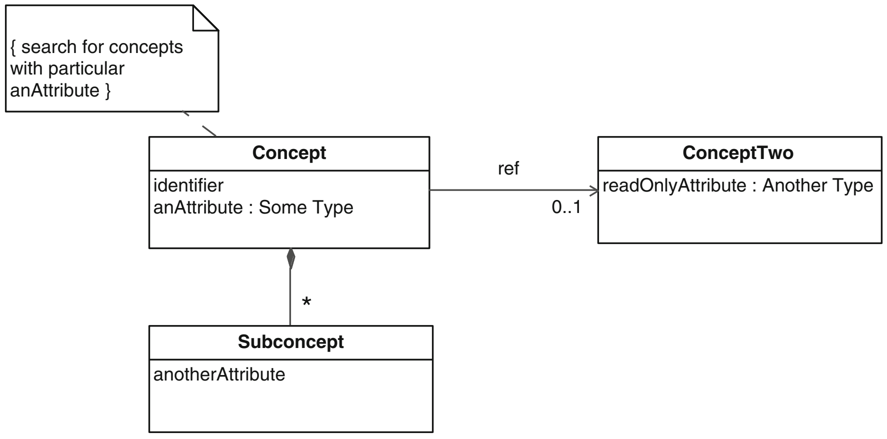


Fig. 11.2 An example domain model

Since content oriented Web services have by definition their value in storing, retrieving and managing content, we assume that a domain model with additional constraints for queries and attribute values can capture essentially enough information to be used as the main source for building the service.

Figure 11.2 illustrates a simple domain model. It has concepts, concept attributes and associations between the concepts. Attributes can have types; associations can have multiplicities (cardinalities) and they can be directed and composite (whole–part relationships). The domain model profile package shown in Fig. 11.1 simply defines the allowed UML elements: classes, associations, attributes, generalizations and comments.

Looking at the domain model, one can outline a respective RESTful Web service: classes look like candidates for resources, attributes as constituents of resource representations, and associations as links. However, it is not obvious how exactly the links are represented, what URIs are assigned to exposed resources, what HTTP operations are allowed per resource and how creation, retrieval, update and deletion of resources is to be allowed.

Resource Model

A resource model re-organizes the elements of a domain model to addressable entities that can be more easily mapped to resources of a RESTful Web service. The concepts of a domain model become resources; depending on their association multiplicities, they either become Items or Containers containing Items. Compositions become resource–subresource hierarchies that are reflected by the URI paths while normal associations become hypermedia references between resources. Attributes are used to generate resource representations and candidates for hypermedia content types. The resource model concept is adapted from Laitkorpi et al. (2009).

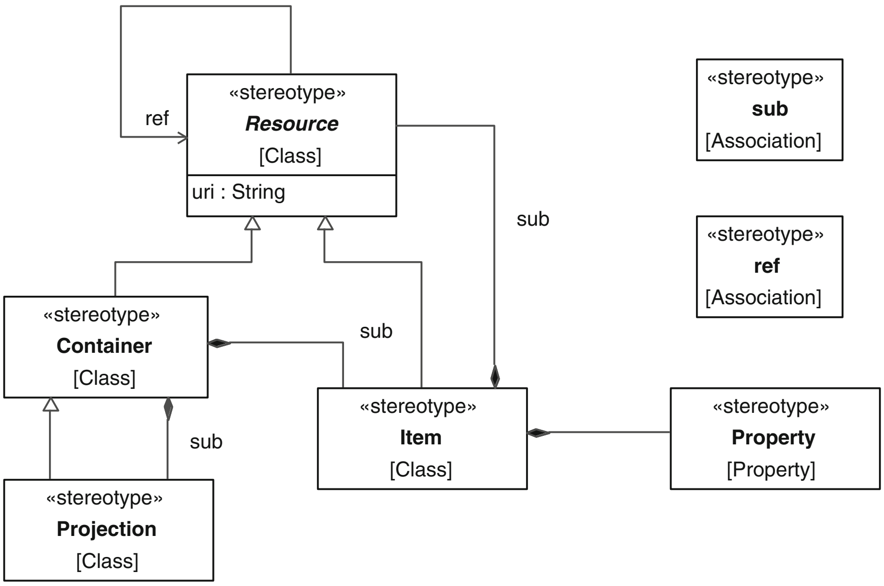


Fig. 11.3 Simple resource model profile

Figure 11.3 shows the concepts of a resource model as a resource model profile. Items represent individual resources having a state that can be retrieved, created, modified and deleted. Containers can be used for retrieving collections of items and creating new ones. Projections are filtered views to containers. Resources can have sub-resources and links to other resources. In addition to the resources and interconnecting links, a resource model instance can be used to infer the other components of a RESTful Web service: resource representations, assigned URIs, and default uniform HTTP operations and response codes. The resource model is formalized into a UML profile, where each element becomes a stereotype.

Representation for Resource r is as follows:

```
<r.name>
  <atom:link rel="self" href="r.uri"/>  # self link
  # for each property p where p belongsto r.property
  <p.name>value of property p</p.name>
  # for each subresource s where s belongsto r.sub
  <s.name>
      # link to a sub resource
      <atom:link rel="self" href="s.uri"/>
  </s.name>
</r.name>
```

In the proposed model, Containers and Projections do not have properties and thus neither do their representations. Each Item i of type T has exactly one id attribute that defines a unique name (among other items of type T). For Containers, id attribute is the name of the container. A relative URI for Resource r is defined as follows:

```
URI(r) = URI(r.parent) + "/" + r.id
```

The default HTTP operations and response codes for resources are as follows:

Item	`GET, PUT, DELETE`	Retrieve, create or update, and remove an item
Container	`GET, POST`	Retrieve collections of items and create a new item
Projection	`GET`	Retrieve a collection of items based on particular criteria

The default status codes for each resource and request can be selected from the HTTP status codes: `200 OK` for successful operation, `201 Created` for successfully creating a new resource, `400 Bad Request` for requests with malformed representations, `404 Not Found` for non-existing resources, `405 Method Not Allowed` for unsupported methods and so forth. The implemented service can decide a finer level of granularity and select more refined communication patterns at will. For example, if a resource contains read-only properties or properties that can only be incremented, attempting to modify them might result in `403 Forbidden` or `409 Conflict` status codes.

Refining a Domain Model to a Resource Model

In the domain model, each concept represents an aspect of the system that will become an addressable resource. Concepts can link to other concepts. Links will become links in the hypertext representations. Concepts can also have subconcepts which are parts whose existence is tied to the parent concept. Subconcepts that do not warrant individually addressable resources can be marked as data types (UML ≪`dataType`≫ stereotype).

Concepts can have attributes that define their state and representation. An ≪`id`≫ attribute represents a textual identifier that will be used when constructing the URI of the concept. For this presentation, we define two additional types of attributes: a ≪`readOnly`≫ attribute is a read only attribute that will be set by the system and that the user cannot modify, and an ≪`appendOnly`≫ counter attribute whose value can only be incremented. The domain model elements can be mapped to resource model elements roughly as follows:

1. Domain classes represent the domain concepts: the content the service is to manage. If not defined otherwise, classes will become ≪`item`≫ resources: addressable resources of their own right with a URI and representation.
2. Attributes of classes as well as data types belonging to classes will become attributes of respective ≪`item`≫ resources and manifest themselves in the resource representations.
3. Associations represent relationships between the concepts and they will become ≪`ref`≫ associations between resources that appear as links in the representations. Bi-directional associations (i.e. associations navigable to both directions) are represented as two directed ≪`ref`≫ associations.
4. Composite associations represent whole-part relationships between resources and subresources, and become ≪`sub`≫ associations between resources.
5. Associations representing collection of elements – i.e. associations having upper multiplicity bound greater than 1 – will manifest themselves as ≪`container`≫ resources containing ≪`item`≫ resources.
6. Notes attached to classes informally describing queries (searches, filtering) are mapped to ≪`projection`≫ resources with attributes for each query criterion. Query attributes that refer to resource attributes are marked with ≪`index`≫.
7. Attributes constrained informally in notes are marked with respective constraint stereotypes. For example, if an attribute is read only (e.g. whose value is to be set by the system), one can use ≪`readOnly`≫. There can be common constraints, but it is usually up to the service architect to identify and pre-define such constraints and how they are mapped to implementation level concepts.

In "Example Web Service: Social Points of Interest" the above approach is applied to a small but non-trivial Web service for social Points of Interest.

Content Negotiation, Inlining and Verbosity

A resource model is independent of any particular content type. In what follows, we will give examples using plain XML. However, it is possible to support any structured representation format for data interchange, including JSON and Atom Publishing Format.

Our framework allows clients to control the number of requests and amount of transferred data through inlining and verbosity. With inlining, the client chooses the interlinked resources to be included to a request response instead of having to fetch each linked resource using separate requests. With verbosity, the client can avoid consuming full representations and get only the most important properties per resource. Both inlining and verbosity are communicated using a custom HTTP headers: `x-rest-inline: linkname, linkname` and `x-rest-verbosity: level`. Inlining and verbosity have proved to be valuable concepts when developing RESTful Web services for constrained clients like mobile devices.

Example Web Service: Social Points of Interest

To exemplify the approach, we develop a simple service that allows users to create and share personal points of interests. Consider the following high-level service description:

> A user can create points of interests (POIs) that can have a title and a description. Users can assign a location (as coordinates) and tags to a POI. Other users can comment POIs and rate them with a thumb up or down vote. Users can search for POIs based on their location (radius and bounded box search), their tags and their popularity (view count). All created content is public, but only the authors of any particular content element can modify and delete them; other users can only read content made by somebody else.

The requirements clearly describe a content oriented Web service: its value is in creating, retrieving, modifying and deleting content, with additional requirements on searching for content and enforcing simple constraints.

Same requirements can yield several similar alternative domain models. Design decisions like whether one resource is a subresource of another one or just linking to each other, or whether a particular property of a resource is promoted to be exposed as a subresource with its own URI, will lead to slightly varying service descriptions. Regardless of the stucture of the particular domain model, however, we can derive a RESTful Web service interface exposing the information content present in the model. If the resulting API turns out to have unwanted or missing features – for example, resources that are always retrieved at the same time but not connected in the domain model, resulting in unnecessary client–server round-trips – the domain model can be adjusted accordingly.

Figure 11.4 shows the domain model for the POI Web service derived from the service description given above. While it is obviously not the only possible model, it nevertheless should be a reasonably good approximation of the requirements.

The domain model defines following structural features:

- User of a system has a unique username and associations to owned POIs, Comments and Ratings.
- POI is owned by a User. It has a title and a description, (preparing for the inevitable future request) creation and modification date, and total number of thumbs up and down. POIs have subelements Viewcount, Ratings, Tags and at most one Location. POI can link to arbitrary amount of Comments.
- Viewcount is modeled as a separate class to emphasize increasing view count being (an implicit) requirement.
- Location has a longitude, latitude and altitude.
- Tag is a simple textual element.
- Comment has a text body and an association to the commented POI.
- Rating has a thumb attribute for thumbs up or down. Each Rating points to exactly one User and one POI.

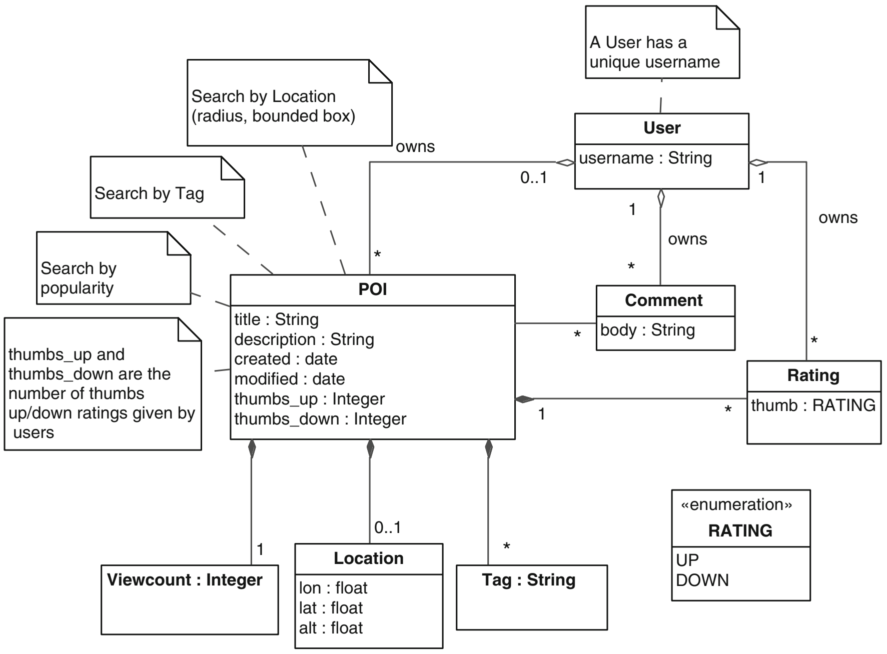


Fig. 11.4 Domain model for social points of interest Web service

In addition, informal features and constraints defined in notes are as follows:

- POIs can be searched based on bounded box, radius, tags and popularity (view count),
- attributes for thumbs up and thumbs down are read only attributes, based on the ratings given for the POI, and
- view count can only be incremented (implicit requirement).

There is no explicit requirement to model Viewcount, Location and Tag as classes; however, a good rule of thumb is to model each concept that might either represent a key concept in the system that we might want to expose as an addressable entity, or that is used as part of a query or whose usage is otherwise constrained.

The use of a composition association (black diamond) denotes that both Location and Tag are parts of a POI and not first class citizens of the Service. This also implies a lifetime constraint: if a POI is removed from the Service, its Location and Tags are removed as well. Further, the multiplicities indicate that any individual Location or Tag element must always point to exactly one POI. There can be at most one Location element associated with any POI, but arbitrary amount of Tags.

In what follows, subsets of the domain model related to specific requirements are looked at in an iterative manner and refined into resource model fragments and RESTful Web service descriptions. The final resource representations, links, and URIs are incrementally merged from partial descriptions.

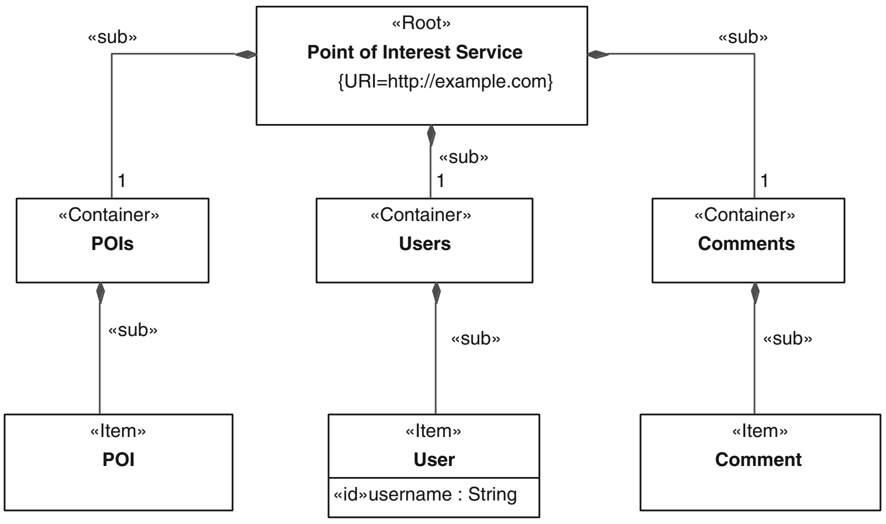


Fig. 11.5 Root element resource model

Root Level Resources

Domain model concepts that are not subconcepts of others but defined as first class entities become root level resources. Resource model for the root level containers is shown in Fig. 11.5. Each class that does not have an owning composite class becomes a root level ≪`item`≫: POI, User and Comment. Unless defined otherwise, we assume there can be an arbitrary amount of each root element and thus create ≪`container`≫ elements for the items. As a convention, we use a plural form of the contained item name for the containers: POIs, Users and Comments. The ≪`root`≫ element is for illustration only, representing the Web service to be built and defining its root URI path. Resources have URIs and thus need an identifier: for containers, the identifier is the container name; for items, we can define one explicitly by an attribute with stereotype ≪`id`≫ (User.username) or omit it and get one generated by the system.

Following the approach defined in "Resource Model," the implied URIs and partial resource representations are as follows (with the "http://example.com" prefix omitted):

```
/pois                            GET, POST
/pois/{poi.id}                   GET, PUT, DELETE
/users                           GET, POST
/users/{user.username}           GET, PUT, DELETE
/comments                        GET, POST
/comments/{comment.id}           GET, PUT, DELETE
```

The operations and HTTP status codes follow from the default operations defined for containers and items. Note that following the above definition, a User is just another resource in the service. In practice, we probably want to restrict creation of new Users to be done through some specific administrative interface. In principle, though, there is no fundamental need why User resources cannot be accessed through the same unified interface. The resource model further implies the following representation fragments:

```
<pois xml:base="http://example.com/">
  <atom:link rel="self" href="pois"/>
  <poi><atom:link rel="self" href="pois/123"/></poi>
  <poi><atom:link rel="self" href="pois/45"/></poi>
</pois>

<poi xml:base="http://example.com/">
  <atom:link rel="self" href="pois/123"/>
</poi>
```

Representations for Users and Comments are derived in a similar fashion.

Points of Interest Resource Model

Resource model for a Point of Interest is shown in Fig. 11.6. The POI, Viewcount, Location and Tag elements of the domain model are refined into respective resource model concepts. POI, Viewcount, Location and Tag elements become ≪`item`≫ elements with respective attributes, with the former being a composite class containing the three latter using a ≪`sub`≫ association. We decide not to expose view count or tags as first class elements with URIs, so we define them as ≪`property`≫ elements. We use two special but generic stereotypes to represent the attribute constraints: ≪`readOnly`≫ for read only attributes (created and modified dates are set automatically by the system) and ≪`appendOnly`≫ for attributes whose value can only be increased (viewcount).

The resource model in Fig. 11.6 implies the following resources:

```
/pois/{poi.id}                    GET, PUT, DELETE
/pois/{poi.id}/location           GET, PUT, DELETE
```

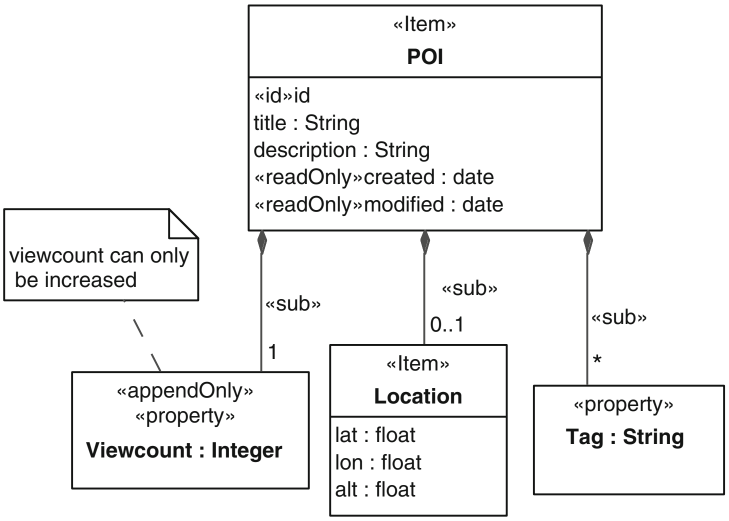


Fig. 11.6 Point of interest resource model and subresources

The resource model further implies the following resource representation with location subresource inlined (`x-rest-inline: location`, see 11):

```
<poi xml:base="http://example.com/">
  <atom:link type="self" href="pois/123"/>
  <title>A title for a POI</title>
  <description>A description for a POI</description>
  <viewcount>132</viewcount>
  <tags>
    <tag>A tag</tag>
    <tag>Another tag</tag>
  </tags>
  <location>
    <atom:link type="self" href="pois/123/location"/>
    <lat>61.4467</lat>
    <lon>23.8575</lon>
    <alt>0.0</alt>
  </location>
</poi>
```

The decision on whether parts of a resource should be promoted as addressable subresources to be individually retrieved and updated is up to the service architect: is a particular aspect of a resource worth exposing independently instead of being just a property of the main resource. In the example, we could choose to expose a ≪`container`≫ Tags containing ≪`item`≫ Tag elements, as well as a ≪`item`≫ Viewcount.

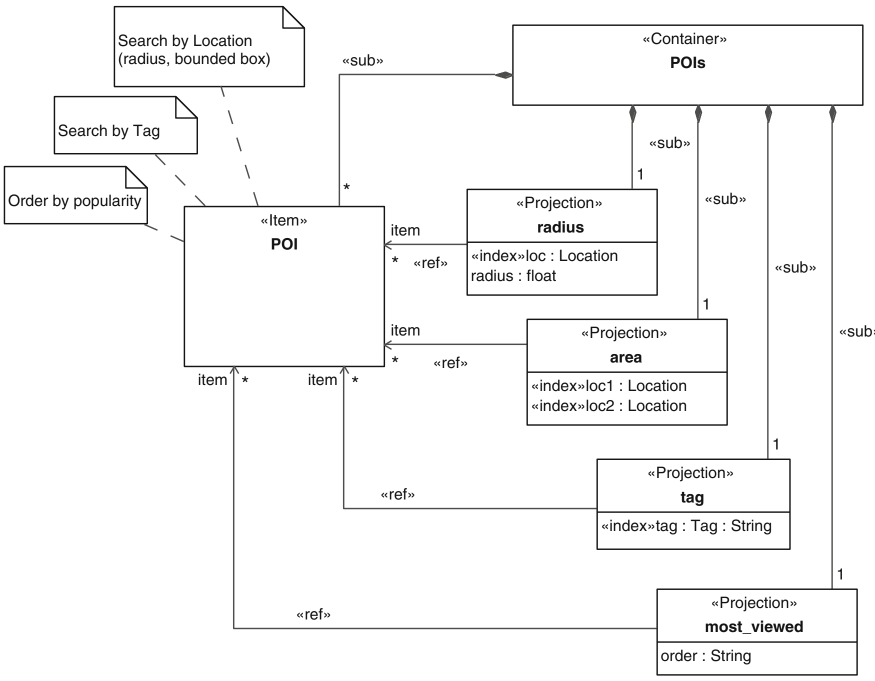


Fig. 11.7 Point of interest resource model for projections: search by radius, area, tags, and view count

Queries and Filtering Resource Model

Following from the domain model fragment shown in Fig. 11.4 and the previous resource models, we further derive the projections corresponding to the different queries. Figure 11.7 shows a resource model for POI related projections. We build a ≪`projection`≫ for each identified query: search by radius, search by bounded box (“area”), search by a tag and order by popularity (“most_viewed”). Each projection is effectively offering a projection to the resources contained by the POIs container, making them are ≪`sub`≫ subresources of the POIs container. As they do not own the resources of their parent container, they instead produce a set of ≪`ref`≫ resource references depending on the query parameters provided by the user. The projection attributes define the search parameters; the ones referring directly to POI attributes can be marked with ≪`index`≫ stereotype to hint later in the implementation phase that the property is used for indexing. Search functionality is obviously critical for the performance of a Web service. Details beyond simple searching over indexed attributes are beyond the scope of the chapter.

The resource model implies the following new projection resources:

```
/pois?q=radius                                  GET
       &loc.lon={poi.location.lon}
       &loc.lat={poi.location.lat}
       &radius={radius}

/pois?q=area                                    GET
       &loc1.lon={poi.location.lon}
       &loc1.lat={poi.location.lat}
       &loc2.lon={poi.location.lon}
       &loc2.lat={poi.location.lat}

/pois?q=tag&tag={poi.tags.tag}                  GET

/pois?q=most_viewed&order={String}              GET
```

One can name the queries as above or alternatively, if a query is uniquely identifiable by its parameters, omit the name. It is possible to combine the queries and have queries with several parameters like:

```
/pois?lon=23.8575&lat=61.4467&radius=0.5&
      tag=restaurant&tag=food&order=desc
```

An example of a POI projection representation is as follows:

```
<pois xml:base="http://example.com/">
  <atom:link rel="self" href="pois"/>
  <atom:link rel="search"
             type="application/opensearchdescription+xml"
             href="pois"/>
  <poi><atom:link ref="self" href="pois/123"/></poi>
  <poi><atom:link ref="self" href="pois/456"/></poi>
</pois>
```

A client can have design time knowledge of the supported query templates, but this couples the client to the service. The URI templates implied by the projection resources can also be mapped to a service description that a client can access at run time. In the above representation, however, we have chosen to use the OpenSearch content type. This way the client can dynamically retrieve descriptions of the currently supported projections for a container.

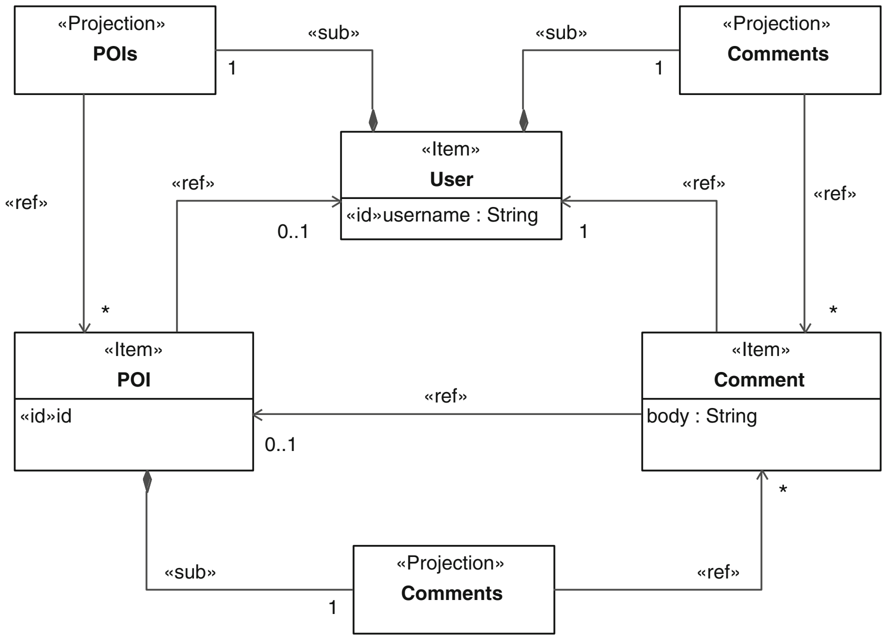


Fig. 11.8 Comments resource model for user and POIs

Comments Resource Model

From the domain model in Fig. 11.4, we can infer that User has a collection of links to POI elements and Comment elements, POI has a collection of links to Comment elements, and Comment has a link to POI and a User. This is formalized in the resource model shown in Fig. 11.8. The collections of links are ≪`projection`≫ elements containing ≪`ref`≫ links to the items. POI and Comment has a ≪`ref`≫ reference pointing back to a User. The bi-directional association between POI and Comment has been broken down to two relationships: a ≪`ref`≫ relationship from a Comment to POI and a Comments ≪`projection`≫ containing a collection of links from POI to Comments.

The implied new resources are as follows:

```
/users/{user.username}/pois                GET
/users/{user.username}/comments            GET
/pois/{poi.id}/comments                    GET
```

Effectively, the new POI and Comment ≪`projection`≫ elements are convenience URIs to the respective containers with the username and POI id as context. For example, the first URI implies a query /pois?user=user.username.

The implied new resource representation fragments are as follows. For User:

```
<user xml:base="http://example.com/">
  <atom:link rel="self" href="users/bob"/>
  <username>bob</username>
  <pois>
    <atom:link rel="self" href="users/bob/pois"/>
  </pois>
  <comments>
    <atom:link rel="self" href="users/bob/comments"/>
  </comments>
</user>
```

For POI:

```
<poi xml:base="http://example.com/">
  <atom:link rel="self" href="pois/123"/>
  <comments>
    <atom:link rel="self" href="pois/123/comments"/>
  </comments>
  <user>
    <atom:link rel="self" href="users/bob"/>
  </user>
</poi>
```

For Comments:

```
<comments xml:base="http://example.com/">
  <atom:link rel="self" href="comments"/>
  <comment>
    <atom:link rel="self" href="comments/453"/>
  </comment>
  <comment> . . . </comment>
</comments>
```

And finally, for a Comment:

```
<comment xml:base="http://example.com/">
  <atom:link rel="self" href="comments/453"/>
  <body>Comment text</body>
  <user><atom:link rel="self" href="users/bob"/></user>
  <poi><atom:link rel="self" href="pois/123"/></poi>
</comment>
```

Ratings for Points of Interest

Ratings are analogous to Comments described in previous section with a few additional constraints. First, there always exists exactly one link from a Rating to a User and POI, and there can only exist one Rating for each POI by a particular User. Both of these are resource state constraints that have to be enforced by the implementation. Second, only the User can see the individual Ratings he has made, change them and revoke them. Other Users can only see the total amount of thumbs up and down in the POIs. Details regarding user management and access control are Web service specific and beyond the scope of this chapter.

The Rating resource model implied the following additional resource:

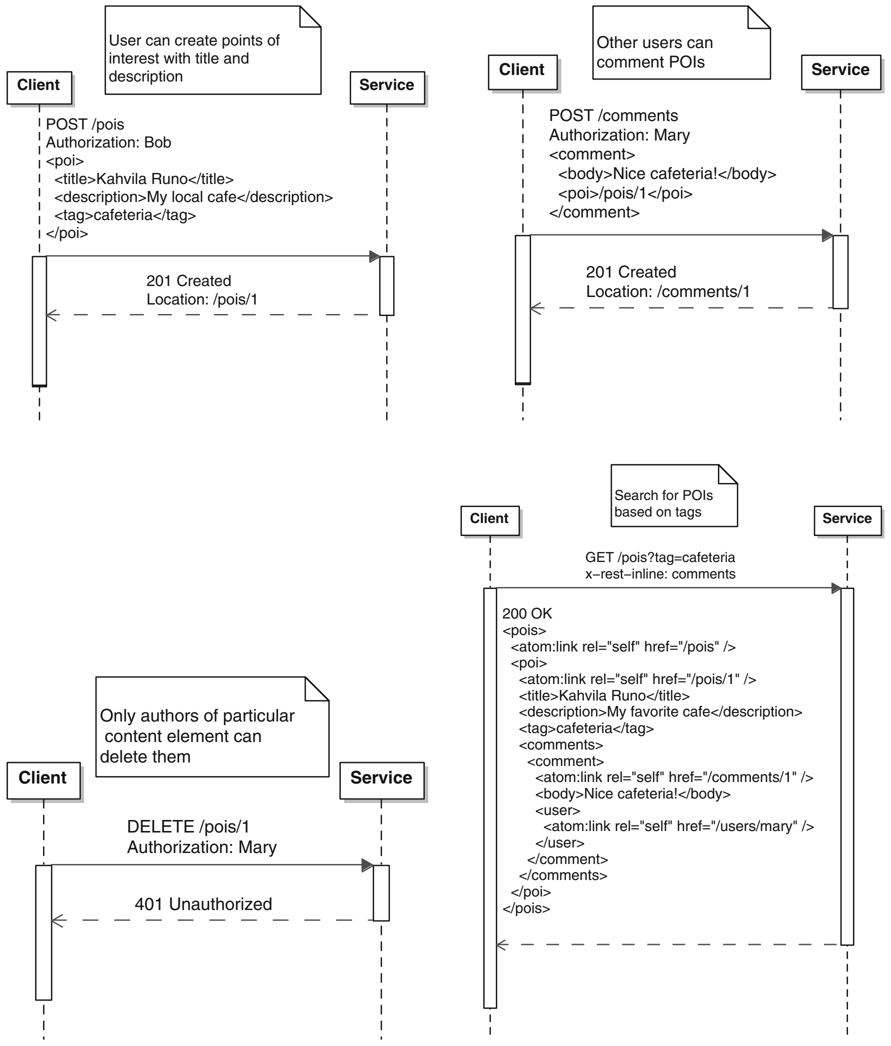
```
/pois/{poid.id}/rating/{rating.id}
```

The representation for a Rating is as follows:

```
<rating xml:base="http://example.com/">
  <atom:link rel="self" href="pois/45/rating/1"/>
  <thumb>DOWN</thumb>
  <user><atom:link rel='self' href='users/bob'/></user>
  <poi><atom:link rel=self' href='pois/45'/></poi>
</rating>
```

Service Requirements Revisited

After the domain model has been refined into a resource model, we want to trace back the service requirements and see how the implied RESTful interface fulfills them. Figure 11.9 shows four example sequence diagrams showing the HTTP level


User can create points of interest with title and description
Client
Service
Other users can comment POIs
Client
Service
Search for POIs based on tags
Client
Service
Only authors of particular content element can delete them
Client
Service

```
POST /pois
Authorization: Bob
<poi>
  <title>Kahvila Runo</title>
  <description>My local cafe</description>
  <tag>cafeteria</tag>
</poi>
```

```
201 Created
Location: /pois/1
```

```
POST /comments
Authorization: Mary
<comment>
  <body>Nice cafeteria!</body>
  <poi>/pois/1</poi>
</comment>
```

```
201 Created
Location: /comments/1
```

```
GET /pois?tag=cafeteria
x–rest–inline: comments
```

```
200 OK
<pois>
  <atom:link rel="self" href="/pois" />
  <poi>
    <atom:link rel="self" href="/pois/1" />
    <title>Kahvila Runo</title>
    <description>My favorite cafe</description>
    <tag>cafeteria</tag>
    <comments>
      <comment>
        <atom:link rel="self" href="/comments/1" />
        <body>Nice cafeteria!</body>
        <user>
          <atom:link rel="self" href="/users/mary" />
        </user>
      </comment>
    </comments>
  </poi>
</pois>
```

```
DELETE /pois/1
Authorization: Mary
```

```
401 Unauthorized
```

Fig. 11.9 Service requirements revisited

interaction between a client and the service: creating a POI, commenting a POI, attempting to remove a POI made by a different user and finally searching for POIs based on a tag with comments inlined.

From Resource Model to Implementation

Resource Oriented Architecture enforces a uniform interface across the Web service implementation. Most of the functionality related to REST HTTP interface,

Table 11.1 Example implementation binding to Java EE, MySQL, Hibernate, and Restlet

	API (Restlet)	Representation (XML/JSON)	Model (Hibernate, Java EE)	Persistence (MySQL)
Item	Restlet resource bound to the URI. Supported default operations are GET, PUT and DELETE.	Representation parsing/generation based on the item attributes. Subresources inlined per request basis.	A native Java object (POJO) generated for each item with a Hibernate Data Access Object and binding to database elements.	Items are rows in respective database table with columns specified by item attributes. References map to foreign keys.
Container	Restlet resource bound to the URI. Supported default operations are GET and POST.	Representation parsing/generation delegated to Items.	Basic retrievals to database, using item mappings.	Containers are database tables.
Projection	Implemented on top of respective Containers.	Representation generation delegated to Container.	Extended retrievals to database, using item mappings.	Stored procedures for more advanced database queries. Tables implied by Container.

representations, models and persistence should be implemented as a cross-cutting concern over the Web service implementation and then applied to the concepts and their relationships defined by a resource model. With proper implementation, it should be relatively easy to extend the service with new concepts in an almost declarative manner.

A concrete binding between the domain model, resource model, and the implementation is done by mapping resource model concepts to the concepts of the selected technology implementation architecture. In our previous work (Selonen et al. 2010b) we have used Java EE, Hibernate, MySQL, and Restlet framework for implementation. The POI service can be implemented in a similar fashion. The binding is summarized in Table 11.1.

Concluding Remarks

According to our experiences (Selonen et al. 2010a,b) engineering RESTful Web services can be difficult for service architects lacking prior experience of REST. The RESTifying approach attempts to systematize the process of moving from service requirements to an implemented service. It applies customary UML domain models to capture domain concepts and together with additional constraints uses them to derive a Web service interface description. It further facilitates capturing the service requirements and communicating them to different stakeholders in a consistent manner with standard software engineering artifacts instead of ad hoc representation examples. The presented approach has emerged from the experiences gained during development of several RESTful Web services. Our goal has been to move away from designing individual APIs to bringing providing access to all content through a uniform programmable Web interface.

The approach has been successfully applied to several service domains that have their core value in storing, retrieving, and managing interlinked content. We argue it is possible to transform any domain model conforming to the domain model profile – i.e. containing only classes, attributes, associations, and generalizations – to a resource model and therefore to an implemented RESTful Web service. Consequently, the approach should be applicable to any service whose requirements can be captured with a domain model and simple constraints on how to access and retrieve content. How to refine a service whose value is in algorithms, processes and complex transactions to resource-oriented, descriptive state content is a valid design problem on its own right.

The RESTifying approach is currently applied manually for designing new services and integrating them with existing Web service platforms. As a next step, we are hoping to experiment with building tool support for the approach: to proceed from a domain model to a resource model and further to a RESTful service description, and integrating the approach to our existing modeling environment. As future research, with proper Web service infrastructure, we hope to be able to generate most of the server side code and client side stubs directly based on the domain model, allowing new resources be added in a generative and declarative way for a consistent Web service interface.

Acknowledgements The author would like to thank Markku Laitkorpi, Petros Belimpasakis, Arto Nikupaavola, Yu You and Vlad Stirbu, the former NRC Service Software Platform team and NRC Mixed Reality Solutions program for their valuable contribution.

References

M. Laitkorpi, P. Selonen, and T. Systä. Towards a model-driven process for designing restful web services. In *IEEE International Conference on Web Services, ICWS 2009*, ICWS, Los Angeles, CA, USA, 6–10 July 2009, pages 173–180.

L. Richardson and S. Ruby. *ReSTful Web Services*. O'Reilly Media, 2007. pages 108–136.

P. Selonen, P. Belimpasakis, and Y. You. Developing a restful mixed reality web service platform. In *Proceedings of the First International Workshop on RESTful Design*, WS-REST '10, pages 54–61, ACM, New York, NY, USA, 2010.

P. Selonen, P. Belimpasakis, and Y. You. Experiences in building a restful mixed reality web service platform. In B. Benatallah, F. Casati, G. Kappel, and G. Rossi, editors, *Web Engineering*, volume 6189 of *Lecture Notes in Computer Science*, pages 400–414. Springer, Berlin, Heidelberg, New York, 2010.

Chapter 12
A Framework for Rapid Development of REST Web Services for Integrating Information Systems

Lars Hagge, Daniel Szepielak, and Przemyslaw Tumidajewicz

Abstract Integrating information systems and legacy applications is a frequently occurring activity in enterprise environments. Service Oriented Architecture and Web services are currently considered the best practice for addressing the integration issue. This chapter introduces a framework for rapid development of REST-based Web services with a high degree of code reuse, which enables non-invasive, resource centric integration of information systems. It focuses on the general framework design principles and the role of REST, aiming to remain independent of particular implementation technologies. The chapter illustrates the framework's capabilities and describes experience gained in its application by examples from real-world information system integration cases.

Introduction

The concept of integration has been present in the software development domain in various forms for the last two decades. Over the years, integration approaches evolved from simple remote procedure calls (Brose et al. 2001) and message passing systems (Monson-Haefel and Chappell 2000) to service oriented solutions and have found their way to become integral parts of programming platforms like J2EE or .NET (Erl 2005). Recent years have witnessed an unprecedented shift in distributed computing towards Service-Oriented Computing (SOC) (Chang et al. 2006), which is gaining prominence as an efficient approach for integrating applications in heterogeneous distributed environments (Erradi et al. 2006). The most popular branch of SOC research is dedicated to advances in Service Oriented Architecture and SOAP Web services (Curbera et al. 2005), but the growing popularity of the

L. Hagge (✉)
Deutsches Elektronen-Synchrotron, Notkestrasse 85, Hamburg 22607, Germany
e-mail: lars.hagge@desy.de

Web 2.0 (Musser and O'Reilly Radar Team 2006) concept has brought increased attention to the REST architectural style as an alternative way of building service oriented environments (Howerton 2007; Vinoski 2007).

Building an integrated software environment in an enterprise often requires developing large amounts of Web services. The integration efforts can be greatly reduced by using a specialized framework for their development. Providing such tools that simplify software development in integration projects is essential for optimizing their efficiency and cost.

This paper describes a framework for rapid development of REST Web services which are suitable for integrating information systems. It first illustrates the application scenario with a simple example, which is used to explain the proposed integration architecture. Then, it introduces the framework architecture, putting particular emphasis on code reusability as basis for rapid development. The next section describes three application examples of the framework, and the final section summarizes experience gained and outlines possible next steps. The paper focuses on the general framework design principles and the role of REST, independent of particular implementation technologies.

Integrating Information Systems Using REST

One of the most important choices to make when building an integration solution is to select an appropriate integration approach and suitable technologies for its realization. These choices can vary depending on the characteristics of the software environment and the particular goals of the specific integration project. This section introduces an example integration scenario and uses it for deriving an architecture for integrating enterprise information systems. The architecture is based on a layer of REST Web services which provide unified access to the information systems of the integrated environment. The section concludes by discussing those types of integration for which REST is well suitable, and those for which it is not.

Integration Architecture

Figure 12.1 (a) shows a simplified information model for equipment documentation. It states that equipments have descriptions in terms of documents, where equipments can be complex items which are built using other equipments, and documentation can consist of various documents with cross-references and dependencies. The schema has to be adapted and specialized for each particular business, yielding an ontology of the target application area. An example is given for facility planning and plant design (b): Facilities are organized into functional subsystems. They comprise functional units, are driven by power supplies, are controlled by safety

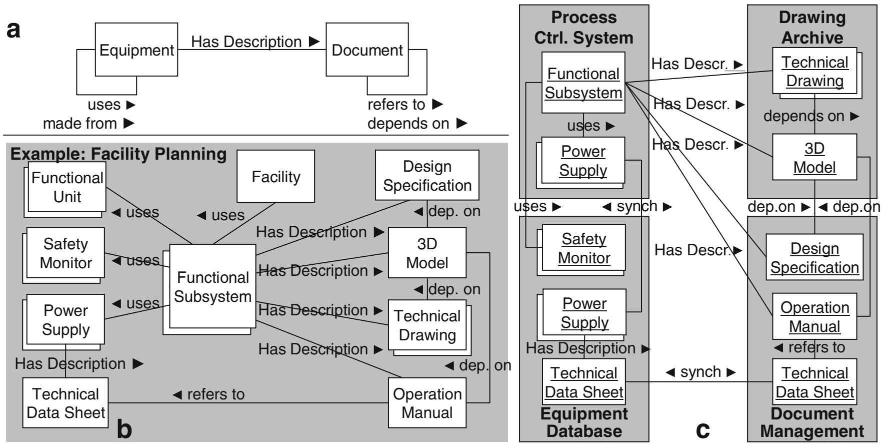


Fig. 12.1 Example scenario illustrating integrated information systems

monitors, etc., all of which are special types of equipments. They are described by a variety of technical documentation, including specifications, design models and drawings, work instructions, and operation manuals, all of which are special types of documents.

When it comes to implementation, ideally a single information system (IS) would support the entire ontology and its business processes, but in practice objects and functionalities are often spread over a number of systems. Figure 12.1 (c) shows a typical example for a deployment scenario: Operators use a process control system (PCS) for setting-up and running of the facility. Technicians use equipment databases (EDB) to keep track of the inventory and organize regular inspections and repairs. Management and staff use a central document management system (DMS) for review, approval and archival, while designers and engineers use a dedicated CAD drawing archive (CDA) for design models and drawings.

The information systems are not independent as there are business requirements which extend beyond the scope of individual systems. Consider the following examples:

1. An operator who may need to respond to an alarm in the PCS, e.g. of an over-heated power supply, would benefit from navigational support to the appropriate operation manual in the DMS.
2. A planned subsystem update, e.g. for improved performance, would require lead engineers to update specifications in the DMS, and then propagate necessary change information to different engineering groups, who then implement the change and update their equipment information and documentation accordingly. The objective would be to coordinate the entire business (change) process independent of any IS boundaries.

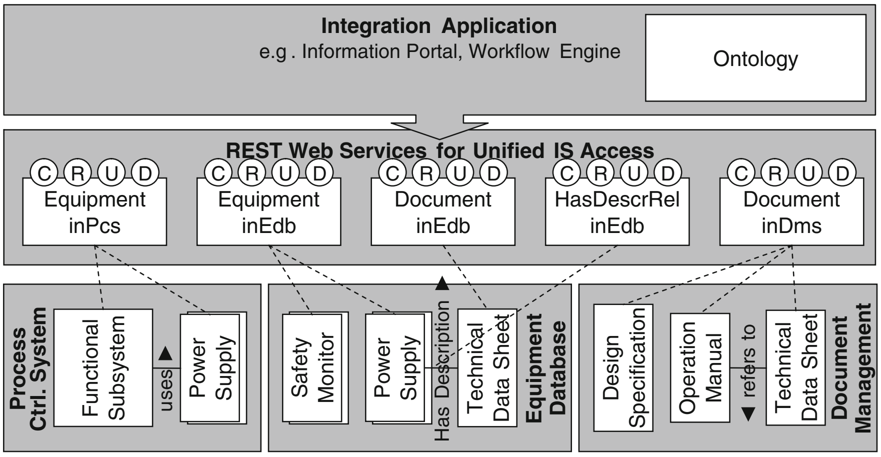


Fig. 12.2 Integration architecture using REST WS for unified IS access

The analysis reveals two characteristic groups of system integration requirements:

- Cross-system item relations: Relations need to be established across boundaries of IS, e.g. Has-Description relations between subsystem equipments in the PCS and technical documents in the DMS and CDA.
- Fragmented objects: Different aspects of the same item are stored in different IS, e.g. technical data are partially kept in the EDB (e.g. parameter values) and partially in the DMS (e.g. signed certificates).

Obviously, the IS environment has to be extended by a component which stores cross-system relations and synchronizes fragmented objects. This could be done by extending the capabilities of one or more of the available IS, or by introducing a dedicated integration component. The latter is preferable as it has the advantage of not interfering with available IS. For interfacing this integration component with the IS, an access layer should be foreseen which abstracts IS access to a uniform interface. Figure 12.2 summarizes the approach, proposing REST Web services for implementing the access layer.

This paper concentrates in the following on the REST Web services which are used for creating the unified access layer.

Identifying and Defining Resources

Integration of information systems based on REST architectural style is resource-centric in its nature as standard REST operations are tightly aligned with the CRUD pattern. Thus, the first step in building an integrated environment involves identifying and defining the resources in that environment. The resource-centric

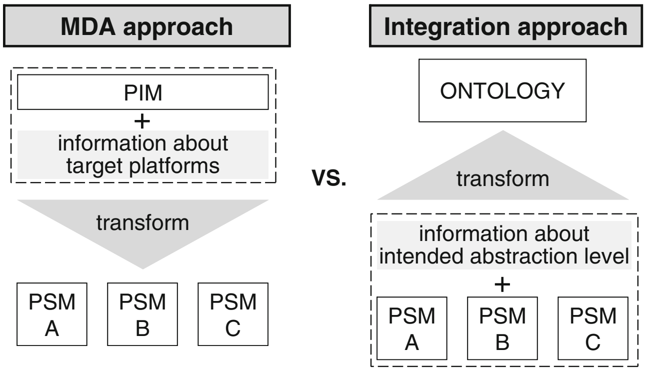


Fig. 12.3 Top-down design vs. bottom-up integration

approach requires a common vocabulary, which contains definitions of all resource types that are used by the participants. Such a vocabulary can be realized in many different forms. One of the most effective ways of formal knowledge representation and sharing it in a coherent and consistent manner among interacting software agents is ontology (Dietz 2006; Guber 1993).

At this point it should be noted, that the initial example can be read in two directions: Figure 12.1 may be the result of a top-down business design process (a–c), which deploys the overall information model to the best-suited available application platforms, or it may be the result of legacy applications which were independently introduced and afterwards brought bottom-up into co-operation (c–a). The former is similar to the MDA-approach (MDA 2003), which transforms high-level platform independent models (PIM) to target systems represented as platform specific models (PSMs). The latter, which is typical for integration projects, implies that the information model would have to be inferred from a bottom-up analysis of the different information systems.

Figure 12.3 illustrates and compares the two approaches. The ontology corresponds to the PIM in MDA terminology, and the models of the individual ISs correspond to PSMs. In MDA, the PIM is the first model to be created, followed by the generation of PSMs for specific platforms (Kleppe et al. 2003). Integration of existing information systems requires reversing this process, i.e. the PIM has to be derived from a set of PSMs of the existing information systems (Szepielak 2007). This involves:

- Comparative analysis of all PSMs to identify similar resource classes in multiple PSMs and ensure they are abstracted into the same concepts.
- Analysis of relations between resources across IS boundaries.
- Analysis of all PSMs to reveal overlapping resource classes and create according cross PSM model mappings.

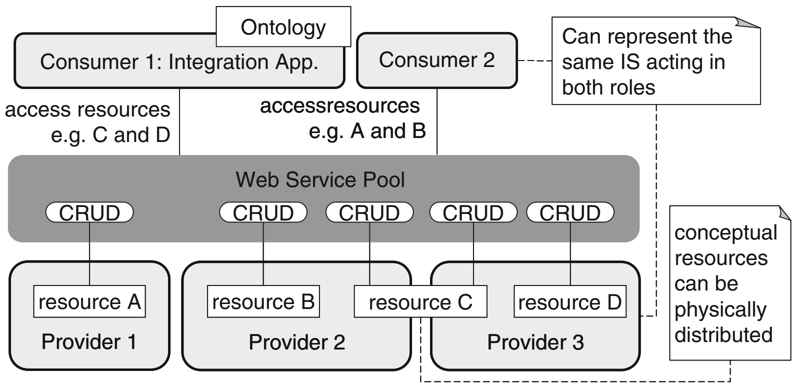

Ontology
Consumer 1: Integration App.
Consumer 2
Can represent the same IS acting in both roles
access resources e.g. C and D
accessresources e.g. A and B
Web Service Pool
CRUD
CRUD
CRUD
CRUD
CRUD
conceptual resources can be physically distributed
resource A
Provider 1
resource B
resource C
Provider 2
resource D
Provider 3

Fig. 12.4 Integration architecture

The emerging ontology has to be checked for consistency and compliance with the original IS data models. It will stabilize in an iterative process. Figure 12.4 redraws the integration architecture from a resource-centric perspective and emphasizes, that ISs can act both as resource providers and consumers.

Modelling Workflow as Resources

The rigorously applied resource-centric approach should completely avoid any form of thinking in categories of processes. All communication among information systems in the integrated environment should be performed with the help of resources only. From a REST perspective, the appropriate way of implementing processes is to represent them as sequences of CRUD operations which are executed on the resources of the ontology.

This approach is feasible for simple transactional workflows. Example 1 from "Integration Architecture", navigating from an alarm in the PCS to the corresponding operation manual in the DMS, could e.g. be written as

1. RETRIEVE status information of Equipment
2. RETRIEVE connected Has Description relation
3. RETRIEVE connected Document

More complex workflows will need richer expressions. Example 2, organizing an engineering change process, could start as

1. CREATE change request
2. APPROVE change request
3. UPDATE specification
4. APPROVE specification
5. …

In this example, approving items denotes that they are read, signed and this way endorsed by responsible persons. Such approvals or sign-offs are common functionalities of information systems, often provided as workflows. To remain compliant with the REST approach, such workflows also have to be represented and treated as resources. This requires translating all functional aspects of workflow into data structure and defining it as an ontology class. For manipulating workflow, the standard CRUD operations can be used with the following interpretation:

- *Create* – start workflow
- *Retrieve* – check workflow status
- *Update* – alter workflow execution
- *Delete* – abort workflow

The above example would then be re-written as

1. CREATE change request (cr-id, title, description, author, ...)
2. CREATE approval workflow (cr-id. reviewer-1, reviwer-2, ...)
3. UPDATE specification (...)
4. ...

If it turns out that reviewer-1 is not available, an alternative reviewer may be assigned by updating the approval workflow. Authors may inquire how many reviewers have already processed the request by RETRIEVing the workflow status, and in case they discover mistakes, withdraw the request for approval by DELETing the workflow.

The described scenario shows how standard CRUD operations can be used to manipulate workflows within an information system.

The scenario neglects that in a "real" business process, the different actions would be conducted by different users. This would require additional steps of routing information to process participants and asking them to perform their actions and acknowledge their completion. Routing, acknowledging, etc. can be treated in the same way as described above for the approval workflow, which leads to the conclusion that any business process can be implemented with this schema.

In case of complex processes, the granularity of resources should be carefully considered. Defining too unspecific resources can lead to insufficient control over a process, while too detailed resources may impose too many actions on the IS users and thus become inefficient. On the other hand, building a library of general-purpose process building-blocks will allow quick and easy future process modification by simply rearranging items in the process sequence.

Suitability of the Proposed Integration Approach

The proposed integration approach has been specifically developed for integrating business information systems. It assumes an existing environment of legacy information systems, which are characterized by transactions-type processing of business objects. In such cases, the strategy of introducing CRUD Web services for accessing business objects and executing transactions is applicable. In other environments, such as e.g. agent-based systems, the applicability of the approach needs to be reviewed.

The effective application of the proposed approach requires careful consideration of the granularity of the information system resources which are exposed as Web services. With the growing number of Web services necessary for intersystem communication, the level of coupling increases, and the environment becomes harder to manage in case of future updates or changes. To avoid the necessity of creating and managing a too large number of Web services, it is advisable to define the ontology on the highest possible level of abstraction that still meets the needs of the IS that need to be integrated. In typical enterprise environments, this condition should be moderately easy to achieve. The proposed method can still be used if fine-grained Web services are necessary, but in such cases it is worth considering if those systems which require such tight integration would benefit from other integration techniques.

As the REST architectural style is highly resource-centric, it is very effective for integrating environments where the primary focus is on data access and manipulation, as representing data as resources is a very straight forward procedure. Environments which focus primarily on complex processes, which span over multiple information systems, are hard to integrate using REST. This is related to the fact that creating a resource interface to a process grows in complexity with the process complexity and the number of involved information systems. While it is always possible to provide access to a complete process through multiple resource interfaces to parts of the process, again the then high number of Web services can lead to tight coupling and according difficulties in maintenance.

While there are situations when the described integration approach is not the best way to proceed with an integration project, it also has its undisputable advantages. First, it allows for a non-invasive integration of existing information systems. The REST Web services that allow for accessing and manipulating information system resources are built on top of the existing code base and have no direct impact on the systems. This ensures that operation of an existing IS (provider) can continue without any interruptions, and users do not experience any side effects to the way they used to work with the system. On the other hand it still allows other IS (consumers) to interact with it through its new interfaces.

Second, in the proposed approach all information systems in the environment can be accessed with the same consistent API, which compared to situations where each IS has its own API greatly reduces development efforts and cost. In particular, the proposed approach allows developers to realize complete integration scenarios without the necessity to learn individual IS APIs, as more IS are added to the environment.

Framework for Rapidly Developing REST Web Services

This section introduces a REST Web services framework with specific emphasis on rapid development. Building an integrated enterprise environment, which often consists of dozens of individual applications that should cooperate, requires developing large families of Web services. The described framework can considerably speed up the development process by achieving high levels of code reuse and easing code maintenance. The section presents the framework architecture, introduces the strategy for code reuse, and describes how the framework is effectively used in large environments.

Objectives

Building an integration solution based on the proposed approach requires developing families of Web services for all resource classes which are defined in the ontology. A development framework shall satisfy the following requirements:

- The framework shall minimize development efforts and time, as the expected number of Web services which have to be provided may be high.
- The framework shall support changing and adapting Web services when the environment evolves; e.g. new ISs are introduced or existing ISs updated.
- The framework shall ensure the Web services are uniform in their structure as far as possible, to make them easier to understand, use and maintain.

As general strategy, the framework shall attempt to achieve an as-high-as-possible degree of code reuse, as code reuse is an efficient and established method of increasing productivity and reliability (Gui and Scott 2006), which significantly accelerates and reduces the development cost of new software (Boxall and Araban 2004). Among other advantages, it:

- Reduces the amount of code to be developed, and thus effort and time.
- Keeps the code base small, and thus eases quality assurance.
- Minimizes code duplication, and thus side-effects of changes.

Strategy for Code Reusability

Reusability is defined as the degree to which a software module or other work product can be used in more than one computing program or software system. The proposed framework facilitates reuse for developing families of Web services by providing code blocks that can be generalized for all or a subset of Web services.

Figure 12.5 illustrates the approach: It shows three Web services, one for retrieving equipment information of power supplies from the PCS, and two for

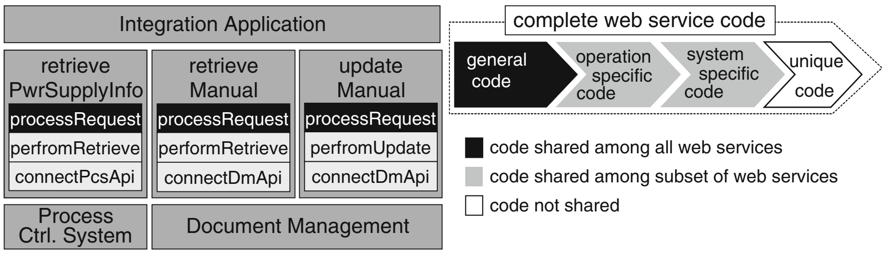


Fig. 12.5 Approach to code reusability

retrieving and updating documents, in particular operation manuals, in the DMS (left). A closer look reveals that:

- The Web services shall provide uniform access to the underlying IS, hence al three of them should respond to similar URIs and provide similar response.
- Two Web services are providing retrieve operations, hence they should have similar internal sequencing.
- Two Web services are serving the same business object, hence they should use the same data definition.
- Two Web services are connecting to the same IS, hence they should use similar calling sequences of the IS API.

Or, more general, the Web services exhibit three major sources of reusability:

- General code that can be shared among all Web services
- Code shared among Web services which are performing the same operation (e.g. code shared between all "update" services)
- Code shared among Web services for accessing a specific IS

The code for a particular Web service can thus be separated into general code, operation specific code, target-IS-specific code, and unique code for that Web service (Fig. 12.5, right). Except for the unique code, the code can be provided by the framework, which is designed to represent each source of reusability by a code block which encapsulates the according functionality:

- A *Request-Response Processor* (RRP) provides functionality which is specific for all Web services, such as request parsing, response formatting and exception handling
- *Operation Controllers* (OC) provide the generic sequencing for specific operations, which can be used for any type of object. E.g. the generic "update" can be written as:
    - open a transaction,
    - find an object using a given set of filtering attributes,
    - if the object exists, update it with a set of new attribute values,
    - on success commit transaction,
    - otherwise, rollback

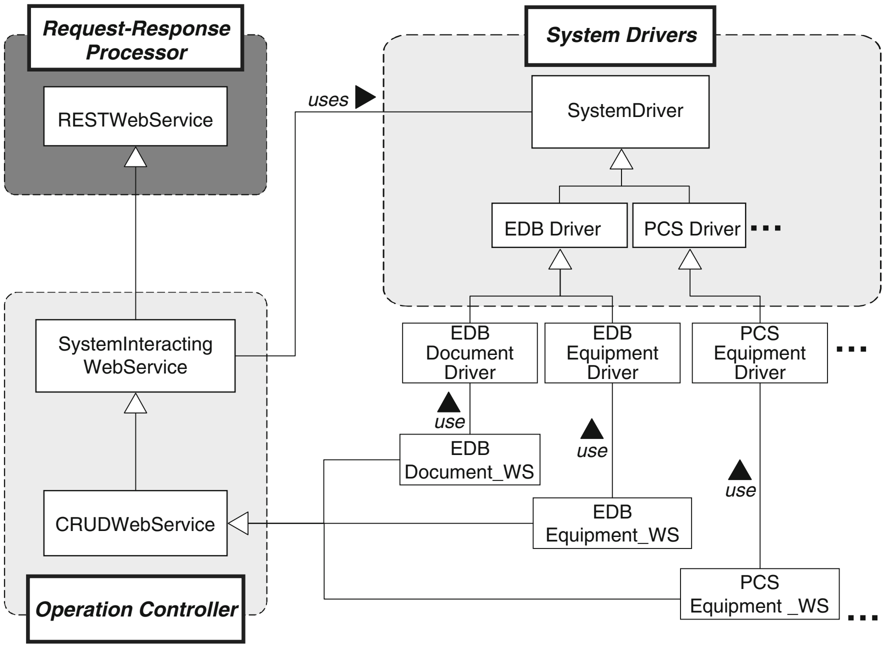


Fig. 12.6 Framework structure

- *System Drivers* (SD) provide functionality for accessing specific ISs, such as connecting to and disconnecting from the IS, beginning and finalizing transactions, and creating, locating, retrieving, modifying and deleting resources. *System Drivers* encapsulate the IS APIs

Framework Structure

Figure 12.6 presents the structure of the framework and the mapping of its classes to functional layers. The left side of the class hierarchy contains classes which implement the operational skeleton of a Web service, while the right side represents system specific code. Figure 12.7 illustrates the interplay of the different classes in an activity diagram. Partitions represent framework classes, and the allocation of actions shows their implementing classes. Structured activity blocks spanning multiple partitions represent abstract methods.

The topmost *RESTWebService* abstract class encapsulates functionality which is common to all REST Web services and realizes the *Request–Response Processor*. The class is responsible for handling incoming requests, processing them and passing their parameters together with stored configuration to the abstract *action()* method for operation-specific processing. Upon successful completion of the operation, the results are formatted and returned. In case of failure, exception handling takes place and error messages are returned.

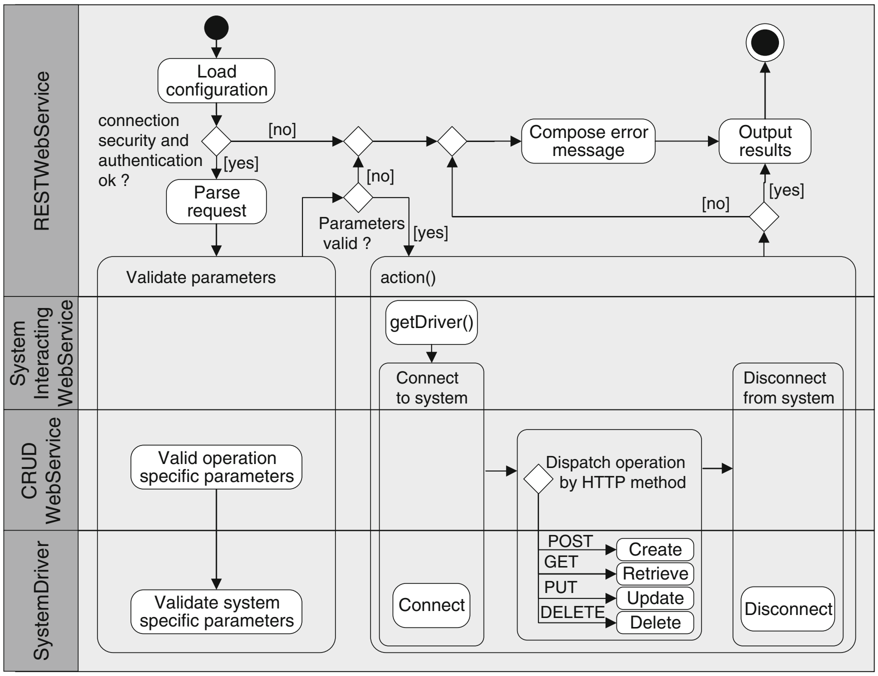


Fig. 12.7 Activity flow and implementation distribution of the framework

The abstract *action()* method is narrowed down in the *SystemInteractingWebService* class, which isolates system interactions common for all Web services. An instance of a *SystemDriver* is created through a call to an abstract factory method *getDriver()* and uses it to connect to the system. Finally, the *CRUDWebService* class performs the dispatching of operations into Create (POST), Retrieve (GET), Update (PUT) and Delete (DELETE) depending on the HTTP method used for the request. Each of the four basic operations is decomposed into a sequence of atomic method calls, which can be overloaded if necessary by an extending subclass.

The *SystemInteractingWebService* and *CRUDWebService* classes realize the *Operation Controller* functionality. In case of operations other than CRUD *SystemInteractingWebService* should be extended by a class that implements a controller for a specific operation.

The *SystemDriver* abstract class acts as a flexible interface for the enclosed set of atomic interactions. It enforces the implementation of basic interactions (like connect and disconnect), but allows for partial implementation of the CRUD functionality and runtime checking of driver capabilities. Such implementation is useful for practical reasons, as not all system resources allow or require the full set of operations. The *SystemDriver* class is a parent class for all specific information system drivers. Single information systems can contain many different types of resources, which require different behavior of the driver at a system specific level.

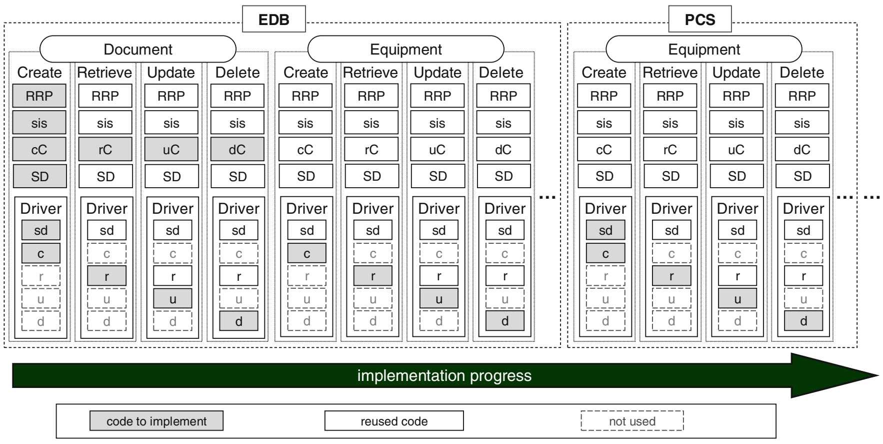


Fig. 12.8 Effective code reuse in Web Service development

Therefore, drivers for specific ISs can have sub-hierarchies of drivers for specific types of resources. The structure of the hierarchy is not enforced by the framework and can be built according to the specification of a given IS.

The actual Web service classes for each of the resource types extend the *CRUD-WebService* class (e.g. *EDB_Equipment_WS*), thereby inheriting the full operational skeleton, and implement the *getDriver()* method so that it produces an instance of a *SystemDriver* subclass appropriate for the particular system-resource combination.

The Framework at Work

The framework structure has been designed to allow for various development strategies depending on the type of a required Web service. There are three basic development paths that are enabled by the framework:

- Rapid development of CRUD Web services for inclusion of new resources to the integrated environment by adding new drivers
- Development of system specific operations not belonging to the CRUD set by extending the *SystemInteractingWebService*
- Development of freeform REST Web services by extending the *RESTWebService*

Figure 12.8 illustrates the rapid development of CRUD Web services for accessing equipment and document information in an EDB, and document in a DMS. Once the first Web service has been completed, subsequent Web services require only minor and well-encapsulated development efforts (Szepielak 2007).

The figure shows that each Web service comprises 6–7 code blocks from the framework, only 1–2 of which need to be newly provided when the pool of Web services is extended. Assuming the code blocks to be of equal size and complexity, this would correspond to 14–33% of code needing to be provided, or an expected average code reuse of at least 70%. This number can get much higher, if the components which have to be developed are small compared to the others.

The different development paths offer developers great flexibility and allow using only partial framework functionality, if required. This way, the potential framework application extends beyond the described integration scenario and allows it to be used for general software development purposes.

Application Examples

The integration approach and the REST Web-service framework have been developed and applied in the engineering data management domain at Deutsches Elektronen-Synchrotron DESY in Hamburg, Germany. DESY is one of the world's leading centers for research at particle accelerators. DESY develops, builds and operates particle accelerators, which are large scientific instruments, and conducts basic research in a great variety of scientific fields, ranging from particle physics to materials science and molecular biology.

This section describes three application examples of the presented REST WS framework: Integrated information access across several information systems, synchronization of information between existing systems, and building new applications on top of an existing environment. The examples involve some of DESY's key information systems, namely:

- The DESY Engineering Data Management System (EDMS), a customized product lifecycle management (PLM) solution
- A combined Geographic Information System and Facility Management System (GISFMS), built with various commercial components
- An Inventory Management System (IMS) based on a commercial IT Asset Management System

Integrated Information Access

DESY has developed a powerful portal which allows users to jointly and intuitively search and navigate the GISFMS and EDMS. The portal provides information about the DESY facilities (buildings and accelerators) through means of metadata querying, hierarchy browsing or visual navigation using maps. The information provided through the portal includes maps and building information from the GISFMS, related with documents and 3D CAD models from the EDMS. REST Web services are used for connecting to the GISFMS and EDMS, querying the systems, and retrieving (lists of) objects.

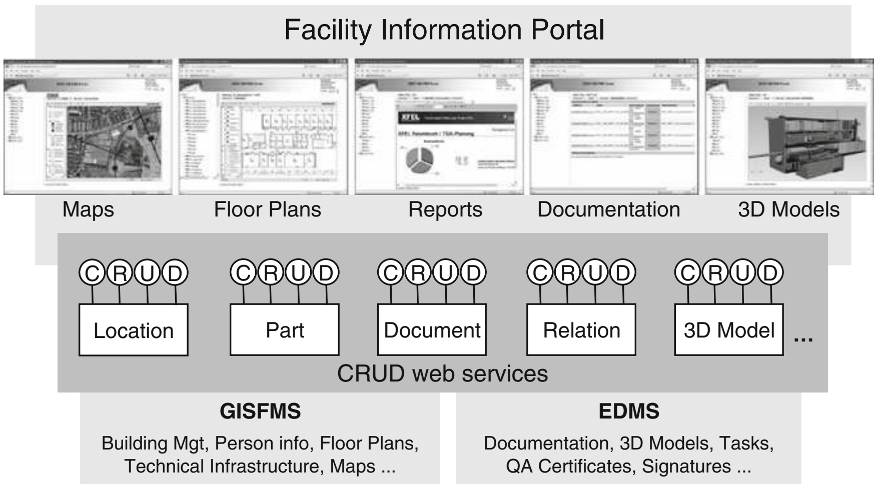

Facility Information Portal
Maps
Floor Plans
Reports
Documentation
3D Models
C R U D
Location
Part
Document
Relation
3D Model
...
CRUD web services
GISFMS
Building Mgt, Person info, Floor Plans, Technical Infrastructure, Maps ...
EDMS
Documentation, 3D Models, Tasks, QA Certificates, Signatures ...

Fig. 12.9 Portal for integrated information access

Figure 12.9 illustrates the architecture of the portal application. The portal provides location-centric information access, i.e. locations are the primary key to information access. For this purpose, the portal provides a tree browser which enables navigating from sites through buildings and floors to rooms, and a map and plan viewer are provided. The Web components retrieve their data from the GISFMS database using CRUD Web services.

The location information of the GISFMS is mirrored and synchronized in the EDMS, where documentation, technical drawings and 3D models are processed and related with their locations. CRUD Web services enable accessing locations, documents, models etc. and traversing relations in the EDMS.

At the time of writing, the portal is already in operation for three years. It serves information for a large-scale accelerator construction project and needs to adapt to growing and changing requirements as the project progresses. So far, it has been both very robust and flexible against changes: Additional information types, such as e.g. 3D model viewing, have been added to the portal without impact on available functionalities, and major software upgrades of the underlying information systems have been successfully carried out without affecting the portal functionality.

Synchronizing Information

A Web-based information system had been developed based on EDMS and IMS for coordinating the installation process in an accelerator project. It registered all the components of the accelerators, provided work lists for the various technical groups, tracked the installation progress, and provided a central information access

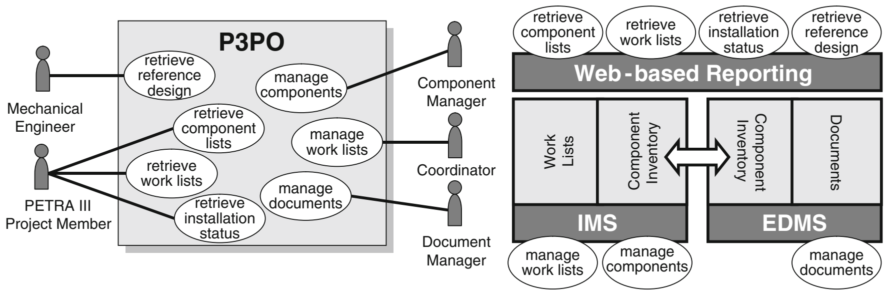


Fig. 12.10 Using IMS and EDMS to support the installation process of a large facility

point for the installation status. The IMS was used for component and infrastructure management and handling work lists, while the EDMS managed the technical documentation of components. An integration component ensured consistency of the information in both systems by propagating changes in one system to the other. The integration component used the REST Web services framework to connect to the systems, access and update objects, and trigger workflows.

Figure 12.10 summarizes the scenario. The different actors are working directly with the ISs, as their roles are mapping 1:1 to one of the systems. Coordinators and process managers use the rich native IMS or EDMS interfaces. The other project workers, who are carrying out installation works in the accelerator facility, are able to retrieve work lists and instructions through a Web-based reporting interface. An integration application in the background ensures that information changes from one system are propagated to the other: If a crucial information change is retrieved from one system, an update Web service is called which propagates the change to the other system.

The application has been realized in very short development time. It was built on top of two information systems, which were in production and starting to show an information overlap. According to the approach, the application has been non-invasive, i.e. did not affect other projects that were also using the EDMS and/or IMS for their activities.

Building New Tools and Applications

The presented framework can be used to build new, specialized clients on top of existing systems. As the DESY EDMS is a very large and complex system, users often request lightweight and easy to use clients for special purposes. The Web services are efficient building blocks for such applications by providing the necessary basic functionalities for connecting, accessing and updating information.

Figure 12.11 illustrates a number of tools and applications, which have been built on top of the DESY EDMS using the REST Web-services framework. They include e.g. direct document searches and accesses from public project Web pages, bulk

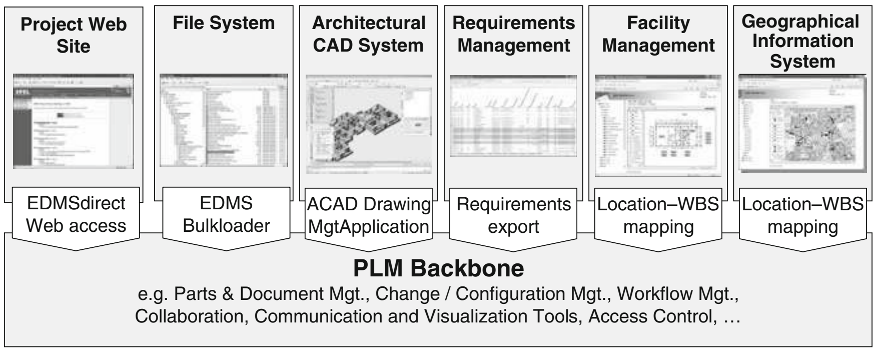

Project Web Site
File System
Architectural CAD System
Requirements Management
Facility Management
Geographical Information System
EDMSdirect Web access
EDMS Bulkloader
ACAD Drawing MgtApplication
Requirements export
Location–WBS mapping
Location–WBS mapping
PLM Backbone
e.g. Parts & Document Mgt., Change / Configuration Mgt., Workflow Mgt., Collaboration, Communication and Visualization Tools, Access Control, …

Fig. 12.11 Special-purpose tools and applications on top of the DESY EDMS

loaders for batch upload of large amounts of files, and connectors for exchanging and synchronizing data with other external databases and applications. Many of these tools are requested at extremely short notice. With the framework in place, such requests can usually be handled.

Summary

This chapter summarizes results and experience from implementing and operating the described framework, and provides an outlook on a strategy for extending the framework architecture for automating the integration of information systems.

Results

The first components of the presented framework are in stable in operation since their initial deployment at DESY in 2005. Numerous extensions and applications have been developed since then, increasing both the scope of operations and the number of accessible information systems.

Figure 12.12 shows the byte code length for the different code blocks of the framework as they have been measured for the initial set of Web services. Figure 12.13 shows the increasing level of code reuse that has been observed as more and more Web services have been developed (Szepielak 2007). The observed level of reuse for all Web services operating in the DESY environment ranges between 83% (for the most complex Web services) and 98% (for the simplest Web services) with an average of 93%. The calculations concern only the internal level of reuse of the framework code itself. Taking into account external libraries used to build the framework, as well as the fact that the Web services are designed to be used in multiple applications, the average level of reuse exceeds 95%.

WS Layer		Web Service Functional Block	BCL
Request-Response Processor		Request-Response Processor (RRP)	11300
Operation Controller		system interaction skeleton (sis)	1680
		create controller (cC)	310
		retrieve controller (rC)	330
		update controller (uC)	280
		delete controller (dC)	270
System Driver		generic System Driver (SD)	1230
		EDMS Driver (sd)	6420
		IMS Driver(sd)	1120
Example resource drivers	EDMS Document Driver	create (c)	1130
		retrieve (r)	4460
		update (u)	1990
		delete (d)	750
	EDMS Component Driver	create (c)	1180
		retrieve (r)	4680
		update (u)	1720
		delete (d)	790
	IMS Component	create (c)	1020
		retrieve (r)	1090
		update (u)	1060
		delete (d)	1010

Fig. 12.12 Byte code length of code blocks in the initial set of Web services

Experience

Setting up the framework was experienced as a time consuming process, but the initial time spent on building the framework resulted in faster and more efficient development of the necessary Web services. The framework allows developing new Web services for accessing further objects from underlying information systems within a few days of work, thus assuring scalability for dynamic environments and increasing integration. The framework also greatly eases the maintenance of existing code.

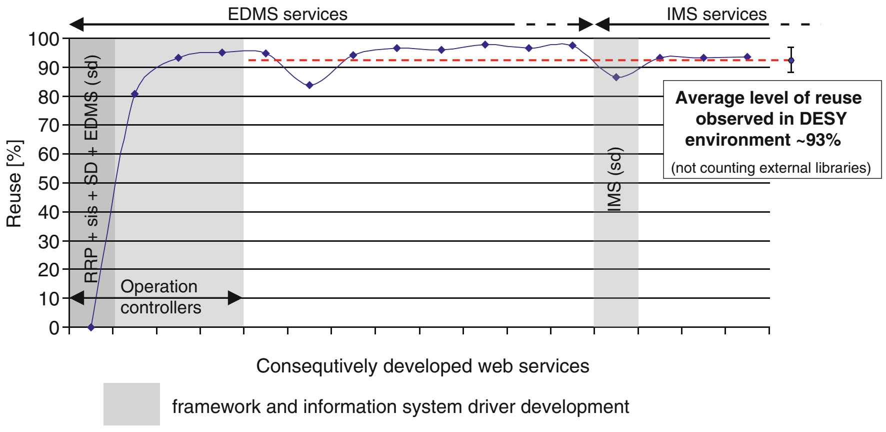


Fig. 12.13 Increasing level of code reuse observed during Web service development

The framework has been built completely from scratch, as at the time of the project no mature enough frameworks were available. With the official JSR for RESTful Web Services in place, JAX-RS (JSR311: JAX-RS 2009), a similar integration framework could be created based on one of the available JAX-RS implementations. Most of the functionality of the RESTWebService and CRUDWebService classes could be taken directly from e.g. Sun's reference implementation of JAX-RS, Jersey. The other classes would still need to be custom-developed, as they are specific to the presented integration framework and to date not available in any generic REST framework.

Several of the underlying information systems have undergone major software upgrades. As the framework successfully encapsulated those systems, no side effects were observed on applications which were built using the Web services. As newer versions of underlying ISs offer richer functionality, some of the Web services may need to be extended to make this functionality also accessible to other applications. In such cases, the REST CRUD paradigm has shown to be well-suited for maintaining backward compatibility and thus avoid impacts on productive environments.

Also the resource-centric approach has shown various advantages in the software development process. The major advantage is that it reflects the business vocabulary, which is particularly beneficial for developers, as they do not need to familiarize with specific system APIs, but can work with a high-level intuitive information access layer which is addressed in the same vocabulary as used in the business itself. This greatly reduces the time until developers get productive and at the same time improves the quality of the resulting software. For example, some of the tools and applications described in "Building New Tools and Applications" have been developed by new staff or students within the first month of their work.

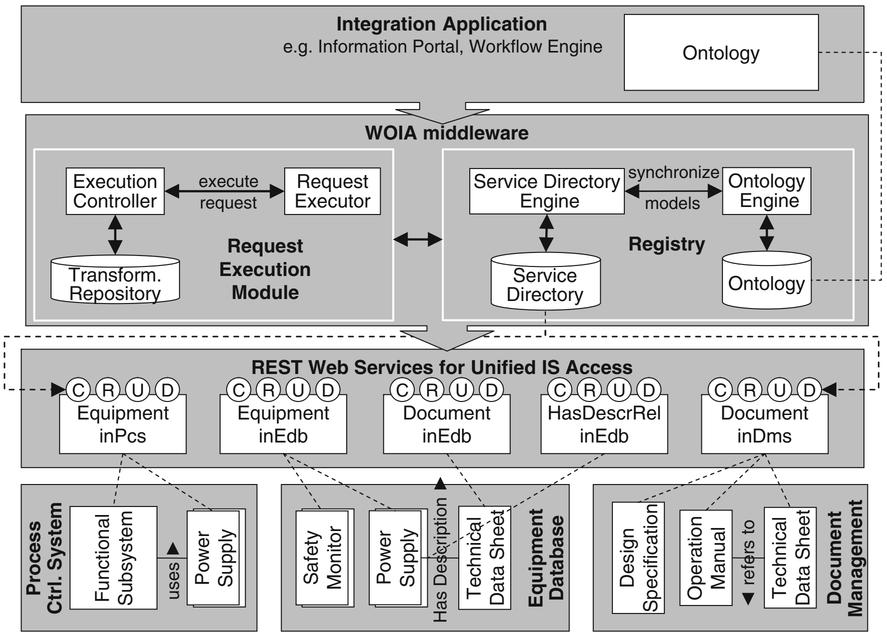


Fig. 12.14 Web-oriented integration architecture (WOIA)

Extending the Integration Framework

The analysis in "Integration Architecture" has shown that all integration applications share two core functionalities: They need to be able to establish cross-system relations, and to handle business objects which are fragmented over several ISs. DESY has developed a dedicated integration application which generalizes these capabilities. It shall act as a middleware which provides Web service registration, discovery, composition and execution capabilities. The architecture which employs this middleware is called Web-Oriented Integration Architecture (WOIA) (Szepielak 2007; Szepielak et al. 2010).

Figure 12.14 illustrates the WOIA middleware in the context of an integrated environment as shown in Fig. 12.2: It consists of a registry and a request execution module, which are both using the ontology to operate. Information systems register within the registry as providers of resources which are defined in the ontology. Based on the registration data, the request execution module allows consumers to directly operate on resources without any knowledge about their providers, almost as if the middleware itself would be providing all the Web services. The middleware has a REST interface which allows the consumers to interact with it in the same way as they would with any other REST service: Consumers send requests for required resources directly to the middleware (the only part of the request that changes is the host name), and the middleware will automatically identify the necessary providers, invoke the required Web services and compose the response, in case of distributed resources by combining responses from several Web services. It also enriches the response with links pointing to the related resources based on the information retrieved from the ontology before the complete response is sent to the consumer.

Using such a generic middleware has the potential to reduce the integration effort to defining an ontology and providing system and resource drivers for the available information systems, while the rest of the required integration software would be provided by the framework.

References

Boxall, M.A.S., Araban, S.: Interface Metrics for Reusability Analysis of Components. In Proceedings of the 2004 Australian Software Engineering Conference (ASWEC'04). IEEE Computer Society, Los Alamitos, CA, pp. 28–37, 2004

Brose, G., Vogel, A., Duddy, K.: JavaTM Programming with CORBATM: Advanced Techniques for Building Distributed Applications. Wiley, NY, USA, 3rd edition 2001

Chang, M., He, J., Castro-Leon, E.: Service-Orientation in the Computing Infrastructure, In Proceedings of second IEEE International Symposium on Service-Oriented System Engineering (SOSE'06), 2006

Curbera, F., Weerawarana, S., Leymann, F., Storey, T., Ferguson, D.F.: Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More. Prentice Hall PTR, Englewood, Cliffs, NJ 2005

Dietz, J.L.G.: Enterprise Ontology: Theory and Methodology. Springer, New York 2006

Erl, T.: Service-Oriented Architecture (SOA): Concepts, Technology, and Design. Prentice Hall PTR, Upper Saddle River 2005

Erradi, A., Anand, S., Kulkarni, N.: Evaluation of Strategies for Integrating Legacy Applications as Services in a Service Oriented Architecture. In Proceeding of IEEE International Conference on Services Computing (SCC'06), 2006

Guber, T.R.: A Translation Approach to Portable Ontologz Specifications. Academic Press, New York 1993

Gui, G., Scott, P.D.: Coupling and Cohesion Measures for Evaluation of Component Reusability. In Proceedings of the 2006 International Workshop on Mining Software Repositories. ACM Press, New York 2006

Howerton, J.T.: Service-Oriented Architecture and Web 2.0. IT Professional, vol. 9, no. 3, pp. 62–64, May/Jun 2007

JSR311: JAX-RS: The JavaTM API for RESTful Web Services available at: http://jcp.org/en/jsr/summary?id=311, accessed on June 08, 2011 (2009)

Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Professional, Reading, MA, USA, 1st edition 2003

MDA Guide Version 1.0.1 available at: http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf, accessed on June 08, 2011 (2003)

Monson-Haefel, R., Chappell, D.: Java Message Service (O'Reilly Java Series). O'Reilly Media, 1st edition 2000

Musser, J. and O'Reilly Radar Team: Web 2.0 Principles and Best Practices. ISBN: 0–596–52769–1 O'Reilly Radar 2006

Szepielak, D.: Web Oriented Integration Architecture for Semantic Integration of Information Systems, PhD Thesis, Silesian University of Technology, Gliwice/DESY, Hamburg 2007

Szepielak, D., Tumidajewicz, P., Hagge, L.: Integrating Information Systems Using Web Oriented Integration Architecture and RESTful Web Services, pp. 598–605, 6th World Congress on Services 2010

Vinoski, S.: REST Eye for the SOA Guy, IEEE Internet Computing, vol. 11, no. 1, pp. 82–84, 2007

Part IV
Application Case Studies

Chapter 13
Managing Legacy Telco Data Using RESTful Web Services

Damaris Fuentes-Lorenzo, Luis Sánchez, Antonio Cuadra-Sánchez, and María del Mar Cutanda-Rodríguez

Abstract Nowadays, companies and service providers have information systems that bring valuable content from both business-support platforms and operation monitoring. The management of this information, used for very different purposes, is usually not reasonably optimized, due mainly to the huge amount of data involved. However, the application of new Web technologies may allow the management of the existing information in a more usable, efficient and dynamic way. This chapter aims to explain the activities to transform an existing collection of data into resources ready to be easily searched and queried, applying advanced Web technologies such as RESTful Web techniques. These technologies have been deployed in this work over traditional tools dealing with services offered to customers, giving as a result a prototype for a telecom company.

Introduction

Due to their economical impact, the management of the information in a company – mainly their services and the huge amount of data collected through them – is a very important issue for business success. However, this is still a very difficult task, due to two main reasons. One of them is that, usually, the information gathered is from different types and different nature. Secondly, companies have millions of customers with different usage profiles, which make even harder the management aspects. In general, providers store valuable operation and business support information on their underlying databases; the managing of this information is not usually optimised.

D. Fuentes-Lorenzo (✉)
Carlos III University, Av. de la Universidad 30, 28911 Madrid, Spain
e-mail: dfuentes@it.uc3m.es

To address these difficulties, tools are needed to ease the task to the company staff. This is the main objective of the SEMNET project explained in this chapter, which applies a new software architecture to the information systems of a telecommunications operator. In the context of this project, our objectives are two-fold. First, the existing information has been accordingly transformed and structured as Web resources to be easily available through RESTful Web services, which can be accessed and manipulated by standard Web-based interfaces, through common programming languages and a common protocol. Second, and to probe the feasibility of RESTifying this existing information, the Web resources have been implemented into a prototype applying a RESTful architecture, to take advantage of scalability and both browsing and data searching facilities.

The remainder of the chapter is organized as follows. "Scenario and Information Sources" covers the scenario where the Web technologies have been applied. "Principles and Approach" explains the approach we have followed to RESTify the legacy data involved. "Prototype" explains the basic implementation aspects of the prototype and depicts some of its functional capabilities. "Related Work" exposes some of the related work and, finally, "Conclusions" concludes with several remarks.

Scenario and Information Sources

The application of RESTful technologies in a certain telecommunication company comes out of the requirements of easily managing the data resulting from one of their network supervising tools. This monitoring facility consists of a set of passive probes deployed within the different networks and services. It allows obtaining information regarding traffic and quality of service parameters, besides customer usage data. Web technologies have been pointed out as the most appropriate to accomplish the task of accessing data in an efficient and easy way, since they considerably improve the de-facto interfaces towards other systems. Besides, Web technologies simplify the presentation of graphical user interfaces for management purposes; standard Web browsers are very well known for the vast majority of the Internet users and tools like Web searchers have been probed very efficient for information retrieval. Due to this reason, a RESTful approach was considered as a good decision to re-model the existing information.

The scenario we have focused on is composed of various available services of IPTV platforms (Television over IP via ADSL access) on the company. The monitoring information extracted from the IPTV platforms consists on Service Detailed Records (SDRs) that contain the most important information exchanged by the user and application servers (video on demand, purchases, Web mail, etc.), from the service request until its end. Each of these records contains the most relevant information of the whole dialogue for a unique requested service between a source and a destination. Information about different services offered and current clients is also managed.

Every piece of generated information is stored in relational database servers where the usability of browsing and searching capabilities can be limited. This stored information is the source entry point for the SEMNET prototype, developed to provide an alternative and enhanced management platform.

Principles and Approach

We explain here the procedures and techniques used to accomplish the RESTification of the legacy data involved.

Architectural Design

As its cornerstone, the prototype design has a set of software-architecture design principles for distributed hypermedia systems based on REST (see Chap. 2). The conceptual improvements that are achieved with the application of REST to the implementation of SEMNET are:

- Clean URLs
- Easy resource discovery
- Variety in response formats
- Easy interoperability among applications
- Scalability

These advantages have identified REST as the more conducive infrastructure to achieve the overall objectives of SEMNET, allowing the definition of a set of resources and operations capable of providing significant advantages over those provided by other SOA implementations.

REST principles are not tied to Web applications, but they can be applied to any distributed system, so it has been necessary to choose a specific REST architecture for the design of the Web-based prototype. The architecture used, considered a subset of SOA, is ROA (see Chap. 2), which combines perfectly with both the design pattern known as *Model View Controller* (MVC) (Reenskaug 1979) – used here as the overall pattern for the design – and the layered architecture described in Larman (2005). Using these design models and architectures has resulted in a dynamic portal easy to maintain and manage.

A general view of the main architectural components can be seen on Fig. 13.1. The information in legacy databases is transformed into RESTful resources, offered to the users through the different controllers in several formats (mainly HTML for human-user requests and XML for computer-user requests). As explained in next sections, users are able to interact with the information through both the prototype system, allocated in a Web application server, and a search engine. As shown in Fig. 13.1, prototype components interact to serve not only user requests (in HTML), but also the search engine results (in XML).

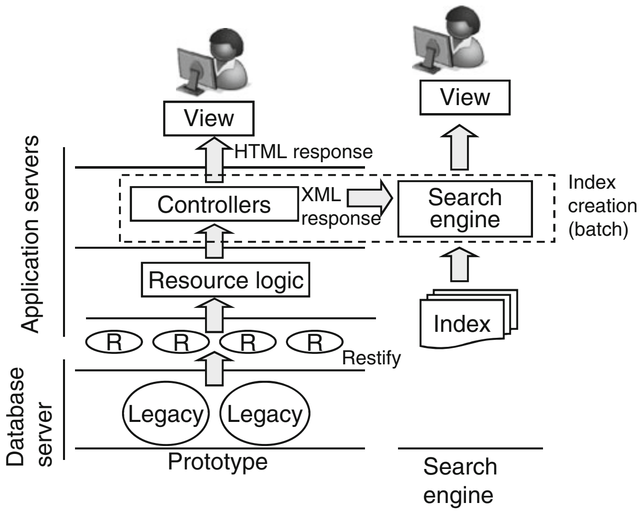


Fig. 13.1 Prototype general architecture

The search engine is able to query the prototype information thanks to the implicit Web services (RESTful Web services) the prototype itself implements; the search engine receives information, in a batch process, by means of XML documents with their associated XML schemas, which are then processed and indexed for future user queries.

Data Representation

The stored information has to be conveniently declared in the form of RESTful resources before an actual system may be implemented. To achieve this, each resource must have its own URL inside the address space. In our case, this URL consists on the type of resource according to the database table it belongs and its key identifier. For this purpose, it is highly recommended that every record of the legacy data has a property (column) defined as its primary key. This way of representing data as a set of unique resources allows the use of generic, uniform and well-known mechanisms for manipulation, exploitation and searching of both information and services. Client interfaces can use just generic libraries and tools (e.g. a browser) compatible with the HTTP protocol to exchange standard messages with the application. Each interaction message contains the necessary information to make the request understandable, so that neither the application nor the user interface needs to recall any state prior to the current interaction.

The information space for the application was formed by legacy data stored in a relational database where the most important tables and their relations are shown in Fig. 13.2. In this example, main data columns and both primary and foreign keys are identified.

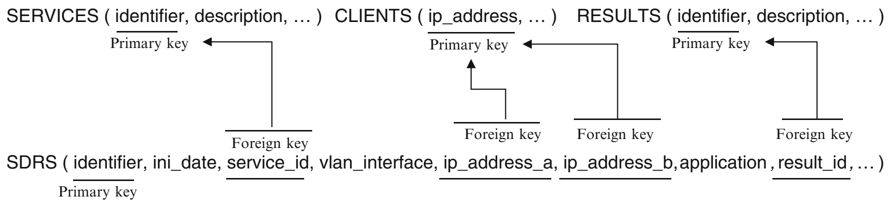

SERVICES (identifier, description, ...)
CLIENTS (ip_address, ...)
RESULTS (identifier, description, ...)
Primary key
Primary key
Primary key
Foreign key
Foreign key
Foreign key
Foreign key
SDRS (identifier, ini_date, service_id, vlan_interface, ip_address_a, ip_address_b, application, result_id, ...)
Primary key

Fig. 13.2 Main relational database tables

The nature (type) of the different columns is diverse. We can mainly find:

- *Temporal information*: Fields such as ini_date, which store the timestamp of the beginning of the SDR session.
- *Connection information*: The IP addresses of the origin and destination involved in the SDR session are available in the ip_address_a and ip_address_b fields.
- *Network parameters*: These parameters represent, for example, the number of kilobytes uploaded and downloaded by the customer, the time required to establish the session or the average response delay to customer requests.
- *Geolocation*: There are fields which represent a text description of the geographical area where the customer is located.

Having this information into account, next subsections explain the main types of resources that have been designed and referenced.

Simple Contents

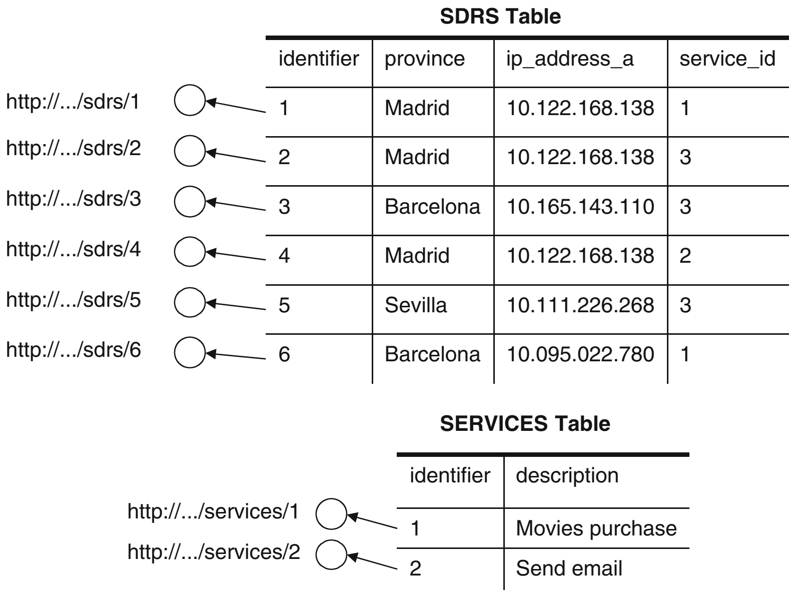
Every information object (a client, a service, an SDR) is included here, where URLs such as http://.../clients/[x], http://.../services/[y] and http://.../sdrs/[z] reference a single client (also known as customer), service and SDR respectively. In this example, x, y and z are the values of the columns which represent the primary keys of the Clients, Services and SDRs database tables. These primary keys are integer numbers in the legacy data used.

Figure 13.3 shows two examples on which URLs are assigned to each of the simple resources involved; as can be seen, each simple resource is a record in a database table.

Complex Contents

This type of resources is formed by collections of simple contents. Following the previous example, the set of customers, services and SDRs are then referred as http://.../clients/, http://.../services/ and http://.../sdrs/ respectively, as shown in Fig. 13.4.

In the figure we can see that two of the complex contents refer to the complete list of SDRs and the complete list of services; every one of these two complex resources has its own URL properly identified.



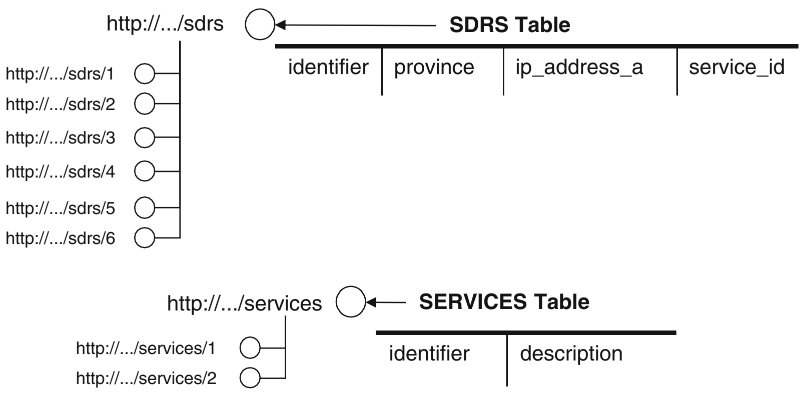
SDRS Table

identifier	province	ip_address_a	service_id
1	Madrid	10.122.168.138	1
2	Madrid	10.122.168.138	3
3	Barcelona	10.165.143.110	3
4	Madrid	10.122.168.138	2
5	Sevilla	10.111.226.268	3
6	Barcelona	10.095.022.780	1

SERVICES Table

identifier	description
1	Movies purchase
2	Send email

Fig. 13.3 Mapping simple contents: From records to REST resources



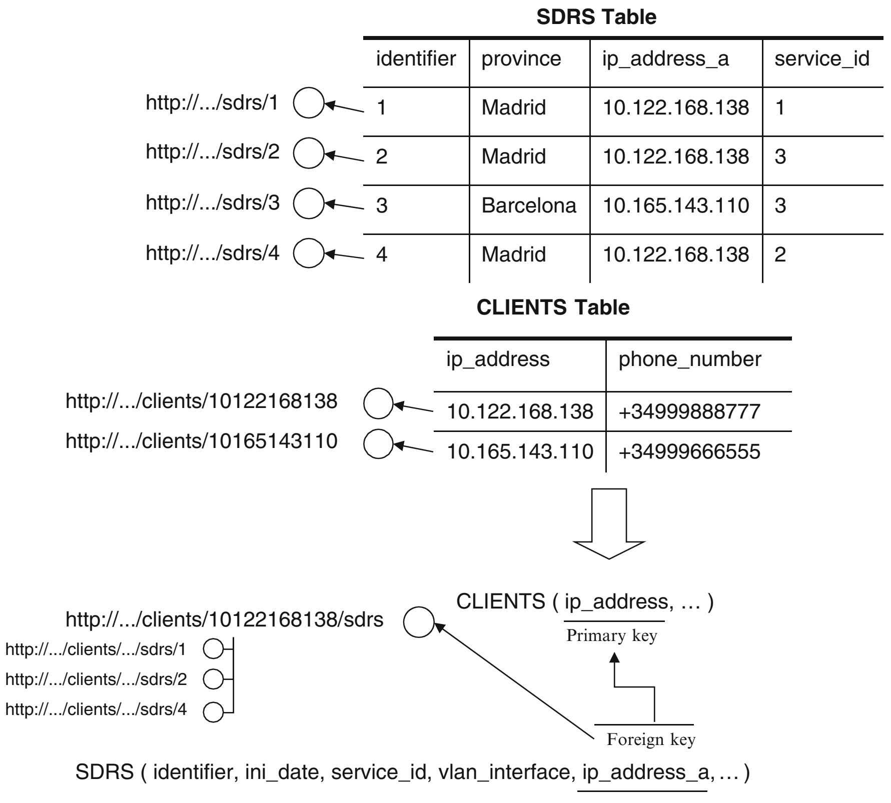
identifier	province	ip_address_a	service_id

identifier	description

Fig. 13.4 Mapping complex contents: From database tables to REST resources

Relations Between Complex Contents and Simple Resources

A simple content can be related with complex contents. In the legacy data used, a customer can have used one or several services and/or originated one or several SDRs; an SDR is related to a service and two customers (origin and destiny), etc. In this case, the relation between the resources is referred as



SDRS Table

identifier	province	ip_address_a	service_id
1	Madrid	10.122.168.138	1
2	Madrid	10.122.168.138	3
3	Barcelona	10.165.143.110	3
4	Madrid	10.122.168.138	2

CLIENTS Table

ip_address	phone_number
10.122.168.138	+34999888777
10.165.143.110	+34999666555

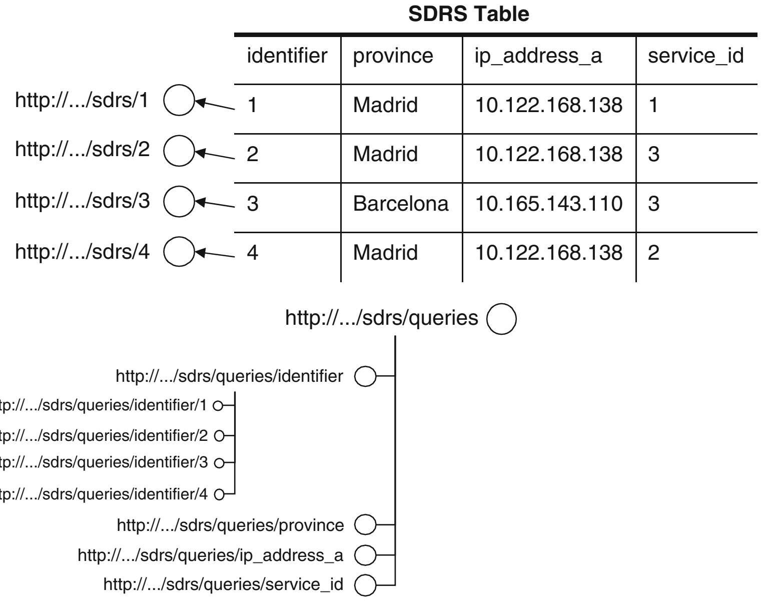
http://.../sdrs/1
http://.../sdrs/2
http://.../sdrs/3
http://.../sdrs/4
http://.../clients/10122168138
http://.../clients/10165143110
http://.../clients/10122168138/sdrs
http://.../clients/.../sdrs/1
http://.../clients/.../sdrs/2
http://.../clients/.../sdrs/4
CLIENTS (ip_address, ...)
Primary key
Foreign key
SDRS (identifier, ini_date, service_id, vlan_interface, ip_address_a, ...)

Fig. 13.5 Mapping relations: From foreign keys to filtered complex resources

http://../[set_1]/[x]/[set_2], where set_1 and set_2 can be any of the database tables involved in any of these relations. x is the value of the primary key in table set_1, representing the simple content.

In Fig. 13.5 we have an example of a relation transformed into another type of complex resource. Given the set of resources of SDRs and clients, the relation between a client and their SDRs can be obtained through the foreign key established in their original database tables. To RESTify this relation, we just have to ask for the complex resource (/sdrs) applied just for a single resource of a complex set (/clients). In the example of the figure, the URL generated by this specific relation points to the set of SDRs where the client is that with the identifier (ip_address_a column) with value 10122168138 (/clients/10122168138).

As shown in the example, a unique resource can have more than one URL which identifies it (the SDR with identifier 1 can be referred as http://.../sdrs/1 or as http://.../clients/10122168138/sdrs/1). However, two different resources cannot have the same URL.



SDRS Table

identifier	province	ip_address_a	service_id
1	Madrid	10.122.168.138	1
2	Madrid	10.122.168.138	3
3	Barcelona	10.165.143.110	3
4	Madrid	10.122.168.138	2

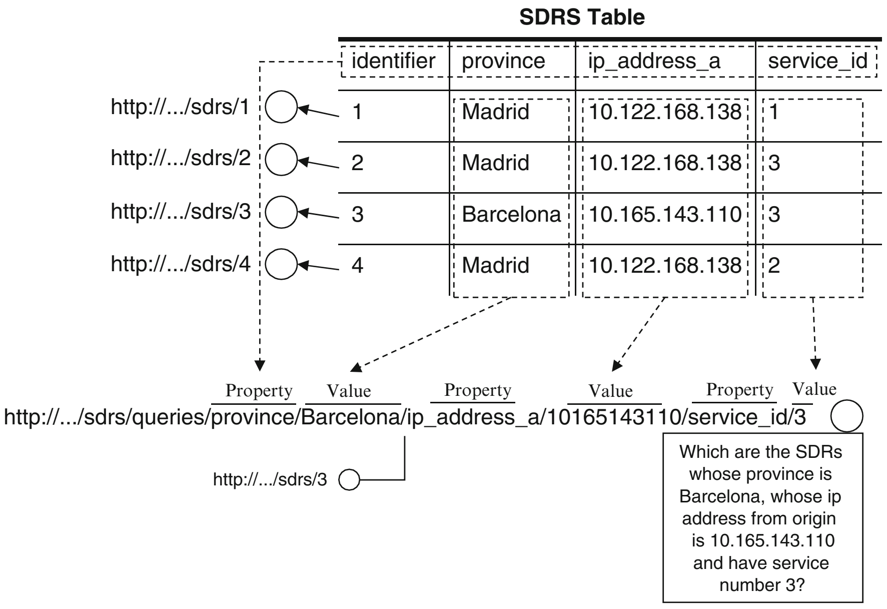
http://.../sdrs/1
http://.../sdrs/2
http://.../sdrs/3
http://.../sdrs/4
http://.../sdrs/queries
http://.../sdrs/queries/identifier
http://.../sdrs/queries/identifier/1
http://.../sdrs/queries/identifier/2
http://.../sdrs/queries/identifier/3
http://.../sdrs/queries/identifier/4
http://.../sdrs/queries/province
http://.../sdrs/queries/ip_address_a
http://.../sdrs/queries/service_id

Fig. 13.6 Mapping definition of queries: From possible properties and values to functions

Functions

Functions represent algorithms applied to a collection of resources. In the scenario presented, the main functions involved deal with querying for a particular set of resources that meet a resource property (a column table) with a certain value, and the generation of statistics graphics. Therefore, the main types of resources we find here and that need a unique identifier are included in the following groups:

- *Definition of a query*: To define a query in a resource collection, the resulting URL has the form http://.../[set]/queries. This URL represents a resource with the possible properties to look up from the set. The set of possible values of a property is another type of resource whose URL has the form http://.../[set]/queries/[property]. Figure 13.6 shows examples of definitions of queries, where possible values of properties are obtained.
- *Execution of a query*: In this case, the URL applied to this feature is http://.../[set_1]/queries/[property]/[value]. Given a result set after the execution of a query, another particular property-value pair can be applied, in the form of the URL http://.../[set_1]/queries/[property]/[value]/[property]/[value], and so on. Figure 13.7 depicts the URL of a query sample.
- *Definition and execution of a graphic*: The execution of a particular service operation can report errors which are also stored in each SDR record. To define a graphic for a specific service, the resulting URL has the form of


SDRS Table
http://.../sdrs/1
http://.../sdrs/2
http://.../sdrs/3
http://.../sdrs/4
Property
Value
Property
Value
Property
Value
http://.../sdrs/queries/province/Barcelona/ip_address_a/10165143110/service_id/3
http://.../sdrs/3
Which are the SDRs whose province is Barcelona, whose ip address from origin is 10.165.143.110 and have service number 3?

identifier	province	ip_address_a	service_id
1	Madrid	10.122.168.138	1
2	Madrid	10.122.168.138	3
3	Barcelona	10.165.143.110	3
4	Madrid	10.122.168.138	2

Fig. 13.7 Mapping execution queries: From properties and values to functions

http://.../service/[x]/graphic, where x is the value of the primary key in the table representing the resource collection of services. This URL represents a resource with the possible dates in the service usage to look up, in such a way that the final URL with the definition of the final graphic has the form http://.../service/[x]/graphic/[year]/[month]/[day]. Figure 13.8 shows the variants of this function resource and their resulting URLs.

Prototype

We explain here the environment and development of the implemented prototype.

Implementation Environment

Implementing a RESTful architecture presents a high degree of independence from development technologies, as one would expect from a SOA implementation. There are many options capable to coexist and cooperate with one another, allowing the correct deployment of RESTful resources. In general, any technology or set of technologies which implement and deploy a dynamic content accessible via HTTP is an option to be considered for the development of a RESTful application.

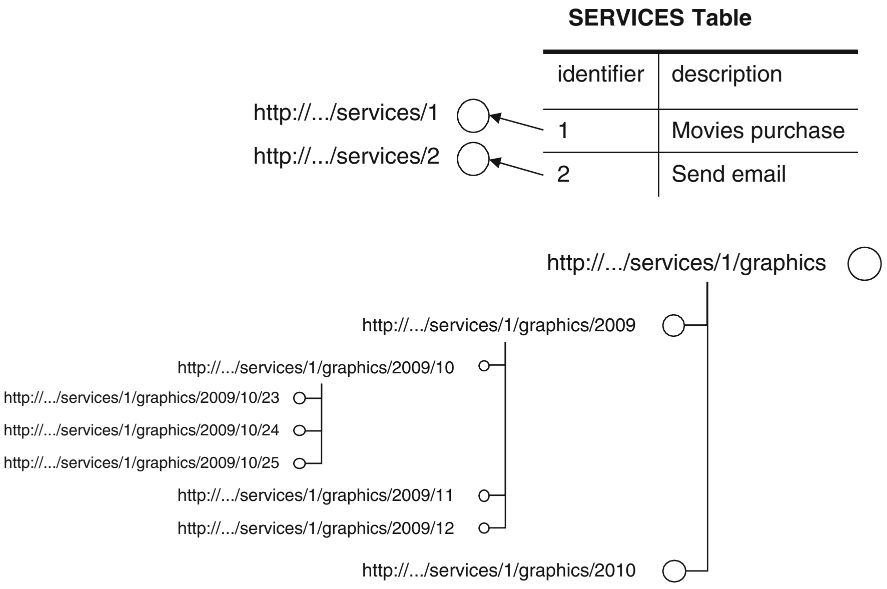

SERVICES Table
identifier
description
1
Movies purchase
2
Send email
http://.../services/1
http://.../services/2
http://.../services/1/graphics
http://.../services/1/graphics/2009
http://.../services/1/graphics/2009/10
http://.../services/1/graphics/2009/10/23
http://.../services/1/graphics/2009/10/24
http://.../services/1/graphics/2009/10/25
http://.../services/1/graphics/2009/11
http://.../services/1/graphics/2009/12
http://.../services/1/graphics/2010

Fig. 13.8 Mapping graphics: From specific data to functions

In this case, the application has been built on a *Rails* (Thomas et al. 2005) environment, upon a Web server. *Rails* is a framework for the development of Web applications. Two of the most important benefits, crucial to the selection of *Rails*, are the following:

- *Rails* implements the MVC pattern automatically.
- *Rails* includes an entire structure to generate RESTful applications and design RESTful resources in a semi-automatic way.

Ruby (Flanagan and Matsumoto 2008) has been the programming language chosen for the implementation. *Ruby*, in addition to being powerful and simple, is also portable, capable of being executed indistinctively on most common platforms including Linux, Windows, Mac, etc.

Figure 13.9 shows the basic implementation flow for a couple of related resource types, the SDR resource collection and the Client collection. After designing the resources needed, and with the help of the *Rails* model layer, the resources designed are implemented in the form of typical *Rails* model classes, called *ActiveRecord* (AR) classes. As can be seen in the figure, an AR object can have relation with another AR object. In the example presented, an SDR is originated by a client, and a client can have generated many SDRs.

Every resource is ruled by a controller with simple CRUD functionality (Create, Read, Update and Delete actions). Every controller is also associated with a view to present the data to the user. The routes the controllers can generate or response to are defined in a configuration file. As the resources are related, the controller of one resource can ask for information to the related resource.

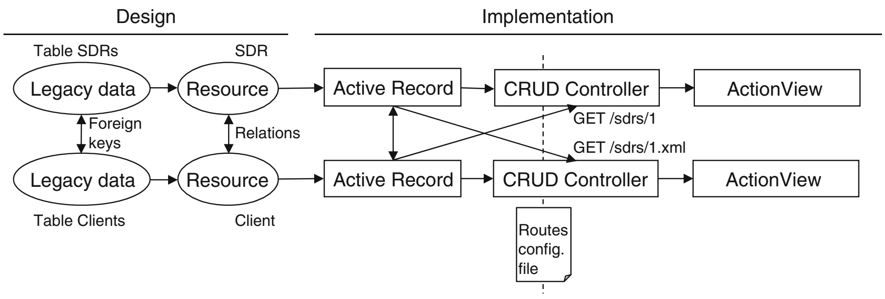


Fig. 13.9 Simple implementation flow of related resources

Functional Aspects

This section shows the main functionalities offered by the prototype, including the search engine. These main functionalities are related with browsing and searching and can be accessed through the GET method of the HTTP protocol.

Browsing

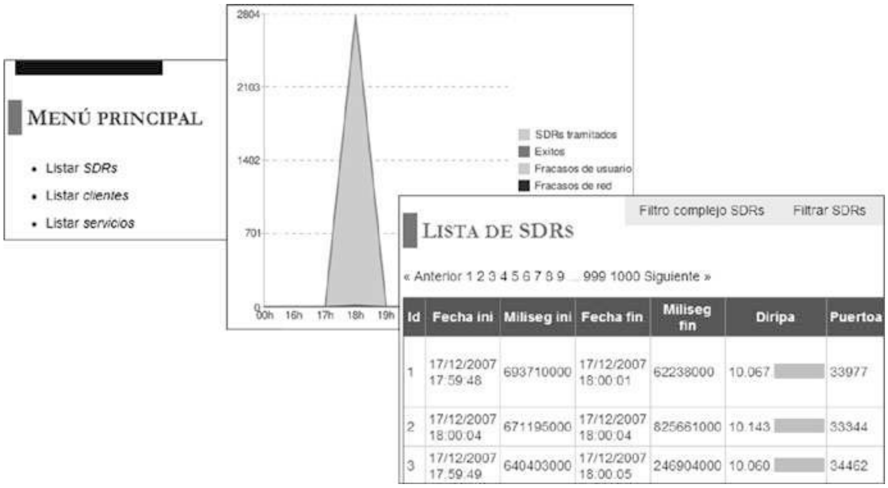
The access to the SEMNET prototype can be accomplished with any HTTP client interface, such as a Web browser. The format in which the requests (and answers) can be made (and received) can be either HTML or XML, although more formats may be added in a simple way.

Users can mainly browse the stored information, which is basically the following:

- Transactions and service-detailed records (that is, the SDRs)
- Requested services on the IPTV scenario (or just services)
- Customers, which represent entities who begin or received the event of a service. Each entity can be a person or a server

The navigation is performed through hyperlinks, without filling in any form to indicate the search parameters or requests, as all of the needed data for the facilities has a unique URL which identifies it. Figure 13.10 shows several screenshots of the homepage or main menu (left), a statistical graph of the use of a particular service (middle) and a resource collection, the list of SDRs (right).

The user can navigate directly to each of these sets of data from either the homepage or from each of every point of information through hyperlinks, due to the fact that the different resources are interrelated and all the parameters needed for a request are in the hyperlinks themselves. For example, the Web page that displays an SDR resource can link to the resource representing the customer who generated such SDR. As shown in Fig. 13.11, step 1, from the list of SDRs, user can see all the details of a certain SDR of that collection.



[redacted]

MENÚ PRINCIPAL

- Listar *SDRs*
- Listar *clientes*
- Listar *servicios*

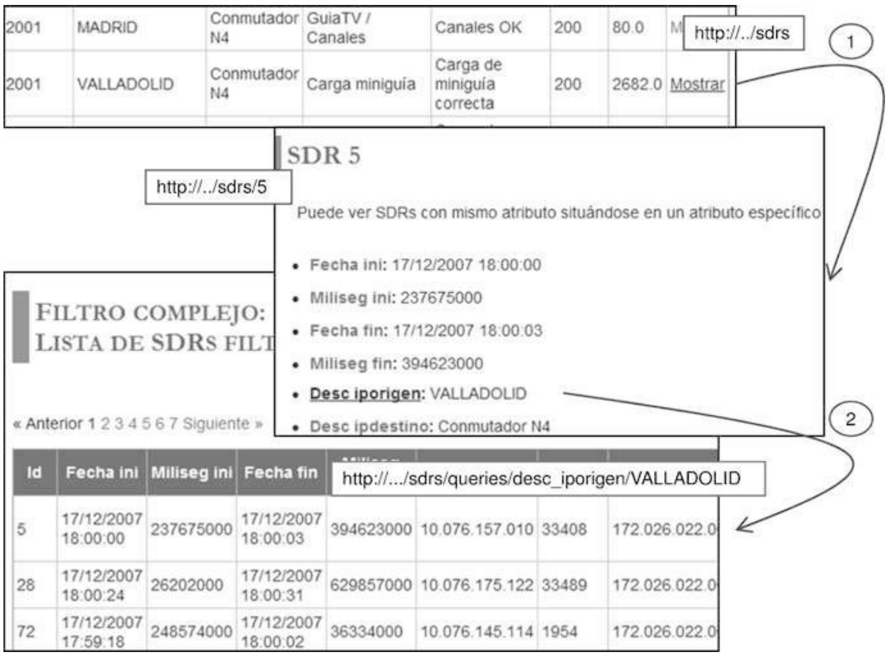
LISTA DE SDRS

Filtro complejo SDRs   Filtrar SDRs

« Anterior 1 2 3 4 5 6 7 8 9 ... 999 1000 Siguiente »

Id	Fecha ini	Miliseg ini	Fecha fin	Miliseg fin	Diripa	Puertoa
1	17/12/2007 17:59:48	693710000	17/12/2007 18:00:01	62238000	10.067	33977
2	17/12/2007 18:00:04	671195000	17/12/2007 18:00:04	825661000	10.143	33344
3	17/12/2007 17:59:49	640403000	17/12/2007 18:00:05	246904000	10.060	34462

Fig. 13.10 Some prototype screenshots



2001	MADRID	Conmutador N4	GuiaTV / Canales	Canales OK	200	80.0	M
2001	VALLADOLID	Conmutador N4	Carga miniguía	Carga de miniguía correcta	200	2682.0	Mostrar

http://../sdrs

1

SDR 5

http://../sdrs/5

Puede ver SDRs con mismo atributo situándose en un atributo específico

- **Fecha ini:** 17/12/2007 18:00:00
- **Miliseg ini:** 237675000
- **Fecha fin:** 17/12/2007 18:00:03
- **Miliseg fin:** 394623000
- **Desc iporigen:** VALLADOLID
- **Desc ipdestino:** Conmutador N4

2

FILTRO COMPLEJO:
LISTA DE SDRS FILT

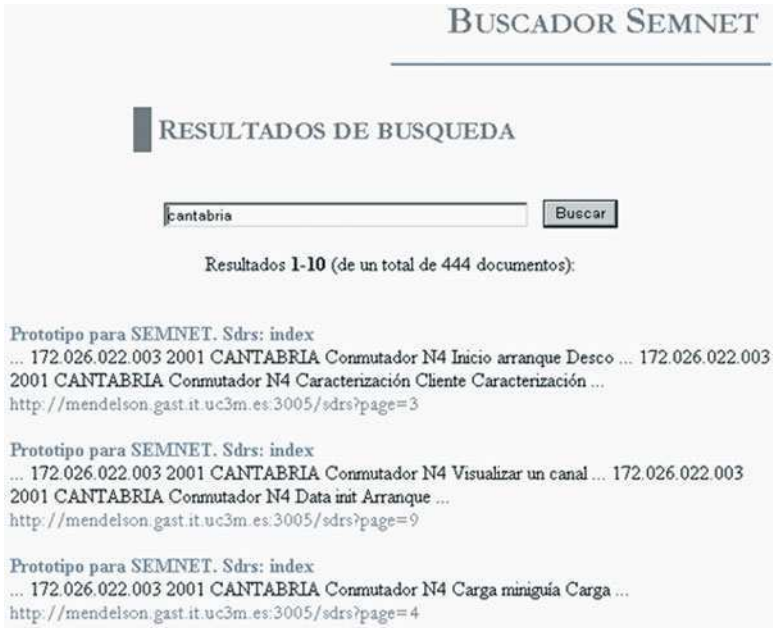
« Anterior 1 2 3 4 5 6 7 Siguiente »

http://.../sdrs/queries/desc_iporigen/VALLADOLID

Id	Fecha ini	Miliseg ini	Fecha fin				
5	17/12/2007 18:00:00	237675000	17/12/2007 18:00:03	394623000	10.076.157.010	33408	172.026.022.0
28	17/12/2007 18:00:24	26202000	17/12/2007 18:00:31	629857000	10.076.175.122	33489	172.026.022.0
72	17/12/2007 17:59:18	248574000	17/12/2007 18:00:02	36334000	10.076.145.114	1954	172.026.022.0

Fig. 13.11 Example of browsing paths among resources

The browsing functionality also provides mechanisms for requesting more elaborated queries. This type of queries (one of the *function* types explained in "Functions") may be applied to certain information based on one or more properties of this data. For example, the user can search for SDRs whose starting data was a



Resultados **1-10** (de un total de 444 documentos):

Prototipo para SEMNET. Sdrs: index
... 172.026.022.003 2001 CANTABRIA Conmutador N4 Inicio arranque Desco ... 172.026.022.003 2001 CANTABRIA Conmutador N4 Caracterización Cliente Caracterización ...
http://mendelson.gast.it.uc3m.es:3005/sdrs?page=3

Prototipo para SEMNET. Sdrs: index
... 172.026.022.003 2001 CANTABRIA Conmutador N4 Visualizar un canal ... 172.026.022.003 2001 CANTABRIA Conmutador N4 Data init Arranque ...
http://mendelson.gast.it.uc3m.es:3005/sdrs?page=9

Prototipo para SEMNET. Sdrs: index
... 172.026.022.003 2001 CANTABRIA Conmutador N4 Carga miniguía Carga ...
http://mendelson.gast.it.uc3m.es:3005/sdrs?page=4

Fig. 13.12 Screenshot of the Web search engine

certain date, or/and the origin was located in a specific region, and so on. To set an example, as shown in Fig. 13.11, step 2, users can navigate from one SDR to a list of the SDRs with *VALLADOLID* (a Spanish city) as the origin, by clicking the hyperlink of the property *desc_iporigen* (the province where the SDR was originated).

Queries information is encoded in the URLs, following REST conventions. This facilitates future searches made by human users, who can also type the query elements directly in the Web browser, or save the hyperlinks as bookmarks for future use.

Searching

The user can also access to SEMNET information through a search engine (Fig. 13.12). This engine incorporates the following components:

- A page-crawler module based on the Nutch (http://lucene.apache.org/nutch/) tool
- A page-indexer module based on Apache Lucene (http://lucene.apache.org/)
- A Google-like interface with a traditional free text box

The availability of the SEMNET information as RESTful resources with well-known URLs facilitates the incorporation of this search system, allowing the user to search information in a simple, versatile and friendly way.

The aim of using a search engine as an alternative access mechanism was to investigate its advantages for querying a corporate database. It avoids the user filling complex forms; the user then just has to enter the query values into the text box. In Fig. 13.12, user searches for SDRs that contain a mention to *cantabria* (a Spanish region).

Related Work

Even though it is difficult to find related proposals in the same scenario or with a related use case as the one presented in this chapter, there are initiatives that already apply RESTful architectures in companies for a better addressability of their resources.

One example is Dogear (Millen et al. 2006), an enterprise-scale social bookmarking system. They also provide design principles referring to online identity, privacy, information discovery and service extensibility.

Practical projects are also developed in Scofield (2008), a book devoted to developers who use Rails regularly for advanced sites and applications.

In Schmidt (2006) or Vinoski (2006), authors focus on the integration of several technologies, including REST, and languages like Ruby to develop the new enterprise applications, embracing the idea of easiness and efficiency these technologies bring to development and maintenance.

In Rosenberg et al. (2008), authors explain the possibility of data integration and composition with RESTful services, a task which may be considered very powerful in the SEMNET context, enabling the possibility of integrate data from different services or network components through Web mashups.

Finally, in Kumaran et al. (2007), authors present the design of a platform for service management with REST. Even though the services they refer to are applied mostly to commerce, they can be extrapolated to any other area.

Conclusions

In this chapter, we have presented the steps to transform the legacy data of a telecom company into well-defined Web resources. All this actual telco information, obtained from one of its distributed network and service monitoring facilities, has been conveniently transformed and structured to be available through RESTful Web services. A proof of concept to validate the approach has also been conducted, in the form of a Web-based application, the SEMNET prototype. This Web application has been implemented to exploit all the previously processed information, allowing different indexing and searching operations over the Web resources obtained.

The model adopted in SEMNET facilitates the recovery, clustering and data mining tasks, simplifying the information integration and dissemination in other systems, and allowing the use of collaborative features (annotation, sharing of bookmarks, reports, etc.) based on Web resources. In addition, this model allows using standard search engines to locate relevant information in a quickly, adaptable and simple way, as it is done on the Internet.

The approach adopted can be easily extrapolated to legacy data of any different business sector. The results also raise a number of tangible benefits to users of management tools, demonstrating that the following advantages applying RESTful techniques can be obtained:

- Web access to information management systems
- Standardising of the information systems interfaces
- Reduction in training resulting from the familiarity with the browsing and searching environment
- Integration into customized interfaces

We have identified some limitations in the potential generalization of this prototype to other information systems; these limitations will allow opening new opportunities within the same spirit of innovation:

- Information persistence: Apart from traditional databases, further techniques for information storage may be needed, to provide a possible expansion of data sources.
- Generalization of network data: There is a need to explore automatic or semi-automatic techniques for the creation of the Web-resources' layers, to avoid the need to perform ad-hoc implementation.
- Knowledge extraction, formalization and exploitation: Knowledge management has not been addressed, but the prototype provides the basis for encouraging this knowledge flow, such as taking advantage of the user experience through search analysis, or facilitating the task of analysing shared information.

References

Flanagan, D., & Matsumoto, Y. (2008). *The Ruby Programming Language* O'Reilly.

Kumaran, S., Li, Y., & Dhoolia, P. (2007). The deep structure of service management. Paper presented at the *ICEBE '07*: *Proceedings of the IEEE International Conference on e-Business Engineering*, 62–70.

Larman, C. (2005). *Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and Iterative Development* (3rd ed.). Upper Saddle River, N.J: Prentice Hall PTR.

Millen, D. R., Feinberg, J., & Kerr, B. (2006). Dogear: social bookmarking in the enterprise. Paper presented at the *CHI '06*: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Montréal, Canada,111–120.

Reenskaug, T. (1979). *The Original MVC Reports*. Oslo: T. Reenskaug.

Rosenberg, F., Curbera, F., Duftler, M. J., & Khalaf, R. (2008). Composing RESTful services and collaborative workflows: a lightweight approach. *IEEE Internet Computing*, *12*(5), 24–31.

Schmidt, M. (2006). *Enterprise Integration with Ruby*. Raleigh, N.C: Pragmatic Bookshelf.

Scofield, B. (2008). *Practical REST on Rails 2 Projects (Practical Projects)* APress.

Thomas, D., Heinemeier Hansson, D., & Breedt, L. (2005). *Agile Web Development with Rails: A Pragmatic Guide*. Raleigh, N.C: The Pragmatic Bookshelf.

Vinoski, S. (2006), Enterprise integration with ruby. *IEEE Internet Computing*, *10*, 91–93.

Chapter 14
Case Study on the Use of REST Architectural Principles for Scientific Analysis: CAMERA – Community Cyberinfrastructure for Advanced Microbial Ecology Research and Analysis

Abel W. Lin, Ilkay Altintas, Chris Churas, Madhusudan Gujral, Jeff Grethe, and Mark Ellisman

Abstract The advent of Grid (and by extension Cloud) Computing along with Service Orientated Architecture (SOA) principles have lead to a fundamental shift in the development of end-user application environments. In the scientific domain, this loosely coupled, multi-tiered software architecture has been quickly adopted as raw data sizes have rapidly grown to a point where typical user workstations can no longer perform the necessary computational and data-intensive analyses. A current challenge facing the design and development of SOA involves the management and maintenance of many loosely coupled service components. As with many large applications, "integration" is equally important as "coding". A resource orientated architecture style serves well in addressing these challenges. Here we present the CAMERA (Community Cyberinfrastructure for Advanced Microbial Ecology Research and Analysis) project as a case study for a SOA in scientific research environments.

Introduction

The advent of Grid (and by extension Cloud) Computing along with Service Oriented Architecture (SOA) principles have lead to a fundamental shift in the development of end-user application environments. No longer do stand-alone applications need to be installed on client workstations. Rather, user applications are now inherently lightweight – relying on remote service calls to "do the work". In the scientific domain, this loosely coupled, multi-tiered software architecture has been

A.W. Lin (✉)
Center for Research in Biological Systems, University of California, San Diego, La Jolla, CA 92093, USA
e-mail: awlin@ncmir.ucsd.edu

quickly adopted as raw data sizes have rapidly grown to a point where typical user workstations can no longer perform the necessary computational and data-intensive analyses.

A current challenge facing the design and development of SOA involves the management and maintenance of many loosely coupled service components. As with many large applications, “integration” is equally important as “coding”. With individual application services written by different developers (often in different programming languages), strict design principles must be followed to ensure a reliable and robust user experience. These principles seek to result in an environment where the user experience appears to be unified, despite a multitude of services working “behind the scenes”.

A resource oriented architecture style serves well in addressing these challenges. Here we present the CAMERA (Community Cyberinfrastructure for Advanced Microbial Ecology Research and Analysis) project as a case study for an SOA in scientific research environments. Specifically, CAMERA is fundamentally based on a collection of REST services. These services are linked together by a scientific workflow environment (Kepler) and presented to end-users in a unified environment geared towards scientific genomic researchers (Sun et al. 2010).

CAMERA

The primary goal of the CAMERA Project is to provide a resource for the scientific genomic community to perform computational and data intensive analysis that would otherwise not be possible with the computational restraints of individual laboratories. These analysis range from data rich with little computational “horse-power” to compute intensive on relatively small amounts of data.

In addition to these computational and data “hardware” requirements, we designed CAMERA to meet two unique needs of the community:

1. *User Driven Analysis*
   CAMERA is a unique resource in that it allows user to design and launch their own custom analysis using CAMERA resources. From a software development perspective this means that services and workflows are developed not only by CAMERA but also by the community at large. Because development does not occur in a centralized, controlled environment, CAMERA’s infrastructure must be both relatively simple and well documented. REST services play critical roles as the limited scope of the constrained create, read, update, and delete (CRUD) interface provides an adequate boundary condition for service behavior.
2. *Provenance*
   Scientific research methods require a full record of transformations applied to data. The adoption of an SOA can make record keeping challenging as many services may act upon a single dataset. A strict separation of concerns (SoC) coupled with REST services makes full provenance recording possible

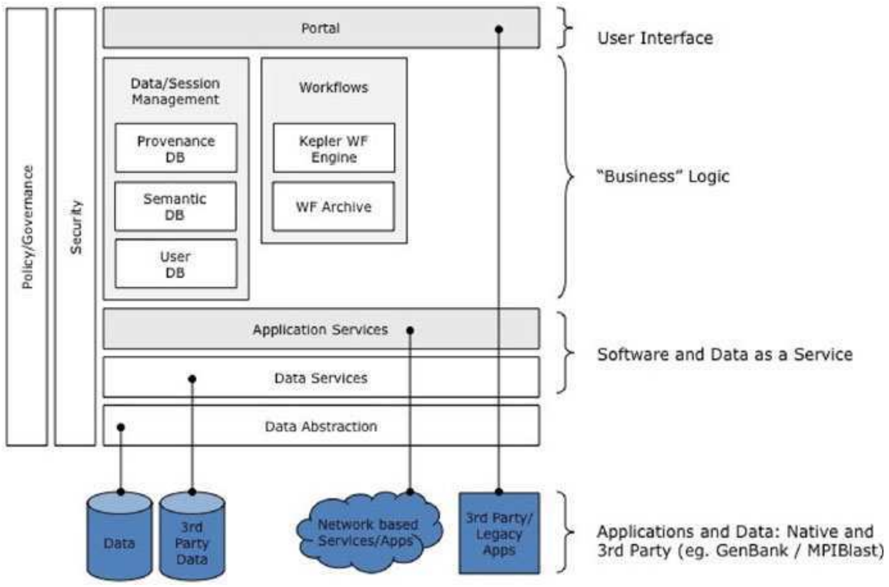


Fig. 14.1 CAMERA 2.0 architecture

in a distributed application environment. Because REST services are state-less the only provenance information that needs to be recorded from the service perspective is version of the service. This SoC allows all state and status conditions to be recorded by the workflow framework and keeps services simple.

CAMERA Resource Oriented Architecture

The layered and modular CAMERA software architecture, as illustrated in Fig. 14.1, is designed to serve two purposes: (1) to provide an adequate SoC for different system components and (2) to allow external scientific developers to create workflows that can fully utilize CAMERA software and hardware resources.

The elements and techniques readily incorporated into CAMERA's architecture include an effective, flexible and intuitive user interface that facilitates and enhances the process of collaborative scientific discovery for domain scientists – accomplished through an end-user interface model that blends both Web and traditional desktop application environments. Primary user interaction is provided via a centralized Web Portal interface. Under the Portal layer are the data and workflow management components to assist with assembly of components into useful and more complex scientific discovery tools. The CAMERA infrastructure currently employs the Kepler workflow system, but it is built for extensions to accept workflows from other workflow systems.

Within CAMERA, Kepler supports the interaction of automated computational tools and human inspection and interaction along with providing capabilities to record the entire processing, i.e., provenance, associated with data brought into the databases. CAMERA also enables users to create and retrieve the processing workflows specific to their own experiments. Query capabilities have been significantly enhanced and the updated approaches engineered into the system now allow users to access data via processing tools hosted by CAMERA. A researcher may thus combine local data and information from outside databases with CAMERA-hosted services and processing tools supported by other groups. With these significant enhancements, the new CAMERA cyberinfrastructure is intended to be more useful, flexible, scalable, and sustainable.

The CAMERA system utilizes project-dedicated, area-dedicated, and very large, multi-community shared resources. These resources span computing, storage, and visualization. Also of note, is that CAMERA has made possible the utilization of data service and abstraction frameworks.

End User vs. Developer

CAMERA is a unique resource in that we have two types of users. The first being what is traditionally thought of as an end user. These scientific researchers come to the CAMERA Portal to utilize analysis and visualization tools for their own research.

CAMERA also has a secondary type of user, the scientific developer. These are the developers that create the tools and services that make up the backbone of the analysis tools used by the end-user. Here, CAMERA's role is to provide a fabric where the developed tools (represented as workflows and services) can be uploaded into CAMERA from the Portal and shared with the greater community.

Design Principles

We implemented CAMERA with a loosely coupled SOA with strictly defined REST services as the underlying layer, which gives us the flexibility to quickly incorporate services from outside of CAMERA. Stateless services handle requests without having to remember any state from one request to the following. Rather CAMERA leverages workflows to maintain state information for clients and to manage persisted information that the services read from and write to the CAMERA data management system. Well-defined, state-less services are critical to the core design. As REST services are inherently state-less, it was a natural design choice.

Within the upper-middleware "business logic" we implemented workflows that link together individual application services. Each workflow accesses one or more services and supplies any necessary logic to maintain state between service requests. In effect, the workflows themselves are the "applications" and deliver the functional process output for the user.

Scalability

Genomic data is growing at an exponential rate. Already, data sizes have out-grown capabilities of standard desktop workstations. User applications now heavily rely on Cloud-based services to perform much of the work. One of the advantages of SOA is how it manages scale. Because CAMERA's core application services are state-less we can, based on demand, provision more instances of the services to be used.

Another aspect of scalability is what we call "developer scalability". Unlike many large-scale projects, the analysis tools are not all developed in-house. Rather, the scientific community contributes a significant portion of the tools. This is where CAMERA's use of REST is most significant.

Most genomic tools were developed prior to the advent of Web services. To integrate them into CAMERA, the first task is to develop a service-based interface. Because REST follows a CRUD interface model, it is simpler for applications developers to add this interface. Even with a simple REST interface, CAMERA further specifies the need for a Camera Service Description Language (CSDL) markup to accompany each service (see pages 324 and 327 for a full description of CSDL and how it is utilized in CAMERA).

Modularity

The importance of modularity increases with the number of components within the SOA. In CAMERA each service is a self-contained component that is state-less, independent and otherwise unaware of other services. This allows us to develop services independently from one another. This adds parallelism from both the computational scalability and development perspective.

Language Independence

Within the CAMERA SOA, client applications use Web services to communicate with each other. While the Web service protocol is an industry standard that is language independent, we have found that in practice it is difficult to have multiple clients and services of multiple languages work together.

In single language environments such as J2EE, SOAP-based services can be an excellent choice as the J2EE framework provides consistency. In multi-language environments, however, we have found the impedance mismatch between SOAP implementation becomes a burden.

Like many large infrastructure projects, however, CAMERA developers spend as much time integrating code as they do developing new code. CAMERA integrates code from a myriad of sources encompassing platforms varying from application server JBoss to simple Perl and PHP scripts. To maintain optimal language independence, the limited CRUD operation of REST was adopted.

Application Services: CAMERA Service Description Language

Within CAMERA, core applications and data are accessed via REST Web services. Principally, CAMERA utilizes RESTful principles to allow the incorporation of software tools for refinement and analysis of community data into workflows. Wrapping community tools for CAMERA takes three main steps:

- Expose low level bioinformatics tools, e.g., NCBI BLAST, as services that are reachable via a unique URI.
- Develop "applications/process workflows" to link together and activate multiple resources via Kepler.
- Deploy portal components to manage multiple "applications" (or to interact with a single resource) and to manage user/application state.

The first step in the process of reengineering these bioinformatics tools into full-fledged workflow elements is the creation of a simple programmatic interface (API) for every tool. Specifically, we expose each bioinformatics tool as HTTP addressable Uniform Resource Indicators (URIs) and parameters so that the application is transported through a simple XML-based data exchange format. This process assumes that CAMERA 2.0 resource services are atomic and stateless. State and session management of their integration in processes are managed by the Kepler workflows and Portal interfaces that are built upon these resource services. While these resources have initially been used only internal to CAMERA as part of the greater infrastructure, they are also being developed towards a goal of enabling third party developers to access and utilize these resources within their own applications.

Example CSDL Document

```
<?xml version="1.0"?>
<CameraWebApp xmlns:camera="http://camera.calit2.net/webapp/wadl">
 <resources base_uri="http://132.239.131.106/camera/rohwer/v1.1/">
    <resource name="circonspect">
      <method name="POST">
        <request>
          <parameter name="u" type="int"
                     descriptive_name="Discard Size" />
          <parameter name="v" type="int"
                     descriptive_name="Trim Size" />
          ...
        </request>
        <response>
          <representation mediaType="text/xml" />
        </response>
      </method>
    </resource>
  </resources>
</CameraWebApp>
```

To accelerate the incorporation of services and resources into Kepler, CAMERA requires a definition language for each service. For SOAP-based services, CAMERA accepts standard WSDL markup. For HTTP-based services, we have specified the CSDL, which is based on the Web Application Description Language (WADL) specification. For more on the usage of CSDL within CAMERA see page 327.

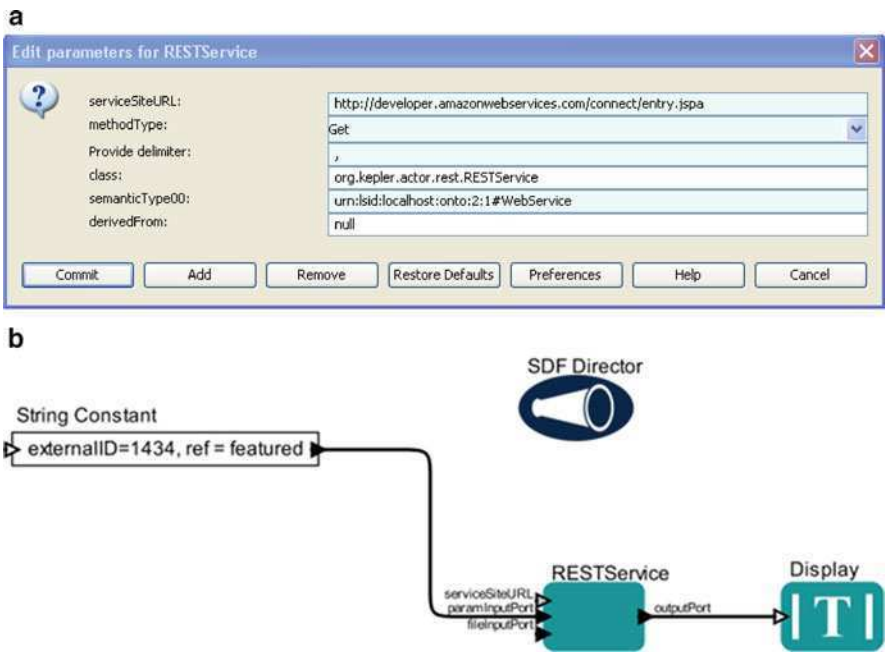
Workflows

While the services within CAMERA are stateless, it is necessary to manage state within an application environment. This business logic task is performed by workflows (Deelman et al. 2009). A workflow is the result of combining data and processes into a configurable, structured set of steps that implement semi-automated computational solutions of a problem. Since their inception, workflows evolved into standard components for many scientific infrastructure projects. The workflow approach offers a number of advantages over traditional scripting-based approaches including the formalization and management of complexity of the scientific process, increased reuse, ease of deployment under different platforms, unified interface for different technologies, provenance tracking, execution monitoring, fault tolerance, and optimization of execution steps.

Within CAMERA, we utilize the Kepler scientific workflow system (Ludaescher et al. 2006). Developed by an open community, Kepler has been used in many eScience projects. While these projects span across multiple scientific disciplines and technical challenges – in particular, the orchestration of Web Services have been a significant focus since the early days of the project.

Kepler workflows are composed of a linked set of components referred to as "Actors" that execute under different models of computation (MoCs). Actors are the implementation of specific functions (i.e., REST services) that need to be performed and communication between actors takes place via tokens that contain both data and messages. MoCs specify what flows as tokens between the actors; how the communication between the actors is achieved; when actors execute (a.k.a. fire); and when the overall workflow can stop execution. The support for multiple MoCs in Kepler is provided by components called "Directors". The designed workflows can then be executed through the same user interface or run from other applications (e.g., the CAMERA Portal).

Finally, Kepler also provides a provenance framework for CAMERA that keeps a record of chain of custody for data and derived products within workflow design and execution. Provenance recording is a very important feature of scientific environment such as CAMERA, as it facilitates tracking the origin of scientific end products, and validating and repeating the experimental processes that were used to derive those products. The Kepler Provenance Recorder (KPR) collects information about workflow structure and executions to enable users to track data generated by


Edit parameters for RESTService

serviceSiteURL: http://developer.amazonwebservices.com/connect/entry.jspa
methodType: Get
Provide delimiter: ,
class: org.kepler.actor.rest.RESTService
semanticType00: urn:lsid:localhost:onto:2:1#WebService
derivedFrom: null

Fig. 14.2 REST actor configuration

domain specific programs. The use of REST services in conjunction with the KPR is critical to the simplicity and robustness of the provenance design.

While a full discussion of the KPR is outside the scope of this chapter, it is a critical component of the CAMERA infrastructure as REST services do not contain and record state nor status information. As a result, we rely on workflows and the KPR to capture all necessary status and other provenance information.

REST Actor

The REST service actor, works with any REST service given that user has a prior knowledge of parameters and files needed to be passed to the service for it to execute the underlying tool. In Fig. 14.2, we configure the REST Actor for a simple service from Amazon Web Services (http://developer.amazonwebservices.com/connect/entry.jspa).

Figure 14.2a demonstrates the REST actor being configured for a Get method, a delimiter (comma in this case) is provided to indentify the different input parameters. From the workflow canvas (Fig. 14.1b) we see two parameters (ExternalID and ref) passed to the service as parameters.

CAMERA REST Actor with CSDL

The REST service actor described above assumes that user already knows what input/file parameters or the service. As previously described (page 323), CAMERA services are described by the CSDL. It contains complete information about the serviceSiteURL, methodType, and the input/file parameters. We have a specific CAMERARESTService actor, which makes use of CSDL file for customization purposes.

As shown in Fig. 14.3, double clicking on the CAMERARESTService actor (Fig. 14.3a) opens up a slightly different a dialog box. Here we provide a URL to the CSDL file (also notice that the delimiter option is no longer available). With the CSDL, the actor automatically configures itself (Fig. 14.3b).

If we now look at the Actor configuration post customization (Fig. 14.3c) we see that that the "serviceSiteURL" and "methodType" parameter are also auto-configured just like other parameters for the service.

Launching Workflows

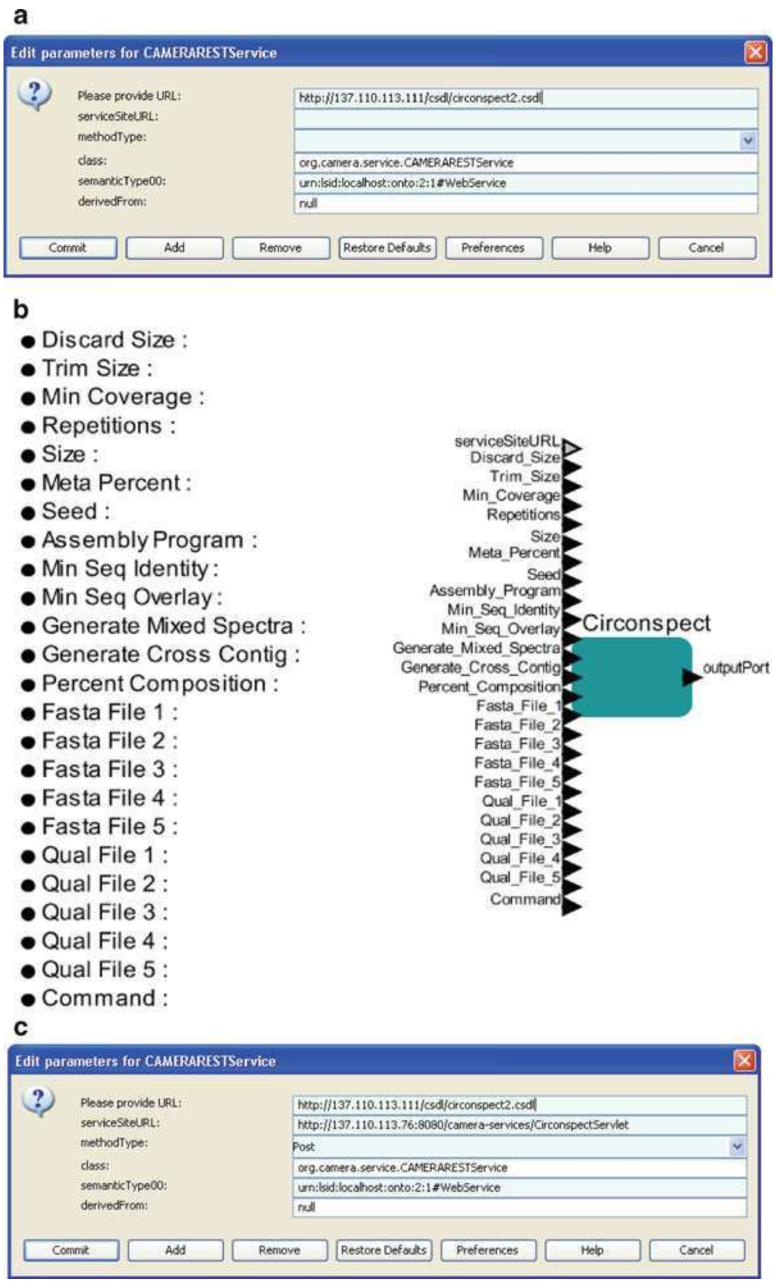
At the heart of CAMERA are Kepler workflows and the services called by those workflows. Workflows are created from a dedicated design interface. After creation they are saved in a XML-formatted Modeling Markup Language (MoML). This MoML can then be uploaded (via the Portal) to run on CAMERA resources.

To satisfy the processing needs of these services and workflows, CAMERA has its own compute cluster. This section describes the CAMERA cluster along with how jobs are run on the cluster.

The CAMERA compute cluster currently consists of 103 Dell 1950s with two dual core 2.33 GHz Intel Xeon processors and between 4 and 16 gigabytes ram and 8 Dell 1935s with two quad core AMD Opteron 2356 processors with 16 gigabytes of ram. This cluster is backed by 18 terabytes of storage on a network file system hosted by a Sun X4500 server. The compute cluster uses ROCKS (www.rocksclusters.org) to manage the cluster, ganglia for statistics, and Intermapper for monitoring. Sun Grid Engine (SGE, www.gridengine.sunsource.net) installation is used to run jobs on the cluster.

Below is a table showing compute hours consumed by processing for the summer of 2010:

Month–Year	Number compute hours consumed
July-2010	99,172
August-2010	88,840
September-2010	96,349


a

Edit parameters for CAMERARESTService

Please provide URL:	http://137.110.113.111/csdl/circonspect2.csdl
serviceSiteURL:	
methodType:	
class:	org.camera.service.CAMERARESTService
semanticType00:	urn:lsid:localhost:onto:2:1#WebService
derivedFrom:	null

b

- Discard Size :
- Trim Size :
- Min Coverage :
- Repetitions :
- Size :
- Meta Percent :
- Seed :
- Assembly Program :
- Min Seq Identity :
- Min Seq Overlay :
- Generate Mixed Spectra :
- Generate Cross Contig :
- Percent Composition :
- Fasta File 1 :
- Fasta File 2 :
- Fasta File 3 :
- Fasta File 4 :
- Fasta File 5 :
- Qual File 1 :
- Qual File 2 :
- Qual File 3 :
- Qual File 4 :
- Qual File 5 :
- Command :

c

Edit parameters for CAMERARESTService

Please provide URL:	http://137.110.113.111/csdl/circonspect2.csdl
serviceSiteURL:	http://137.110.113.76:8080/camera-services/CirconspectServlet
methodType:	Post
class:	org.camera.service.CAMERARESTService
semanticType00:	urn:lsid:localhost:onto:2:1#WebService
derivedFrom:	null

Fig. 14.3 REST actor configuration with CSDL

CAMERA schedules jobs using a “fair” scheduling scheme, or in other words, where job priority is inversely proportional to the amount of processing time consumed by the submitter previously. CAMERA does this by assigning each CAMERA user a project in SGE and giving each project a slice of the system using SGE share tree policy.

In addition to the “fair”, scheduling CAMERA also limits each user to four running workflows at a time. This is done to lessen the impact a new user submitting jobs has on the system, since for new users the scheduler tends to over compensate by giving that user a lot of processing time. Another benefit from this limiting is to prevent any user from dominating the system with long running jobs since a lot of the processing cannot be preempted once it is started without causing workflow failure or a restart of the processing.

Overall the cluster has worked well for CAMERA with the only issues being that SGE tends to have a very low tolerance for file system issues that causes SGE to immediately fail a job. To deal with this issue CAMERA developers have had to modify the applications that submit jobs on the cluster to detect and resubmit failed jobs.

Workflows are a critical component to the CAMERA design. Workflows provide a mechanism to track state and session and also provide an avenue for scientific developers to contribute their own tools and analysis into the system.

Challenges

In design of an SOA, RESTful or not, comes with challenges that are different from stand-alone applications. With a multi-layered architecture, a central challenge is killing jobs that may have processes that span across all layers. In addition, SOA adds challenges to proper testing and deployment of applications, both due to the sheer number of components that must be managed and to the fact that services utilized that may be outside of our immediate purview.

Killing Workflow and Services

We also had to consider the pit falls of SOA, in particular with respect to computing and find ways to address them. From the CAMERA Portal, each job launch starts a specific workflow based on the user’s selection of input parameters and files. Most genomic data (e.g., fasta file) consists of tens to hundreds of thousands of sequences, from which a typical workflow computation lasts anywhere from several hours to several days.

With such computational intensity, it is imperative that CAMERA allows users to terminate jobs. As with most SOA, it is not enough to simply terminate the “parent” process (in our case the workflow) as that in most cases does not also terminate the remote processes. As such, we have devised a mechanism to address this problem. Since Kepler is launching the Web services through the workflow it was deemed the workflow's responsibility to stop those services by passing an appropriate terminate signal when a request is sent to end the workflow. This mechanism is developed with "observer pattern" software design principles.

One of the challenges facing the service users is how to terminate the service in case of an inadvertent or capricious launch. Based on the our updated design principles, all the services will accept an additional command "terminate", which in conjunction with job identifier will allow the service to stop the job. This additional command will offer us a way to enable users to save computation resources by terminating the unintended job launches.

However, in our case the design is complicated by the fact that we have three distinct entities to deal with (1) CAMERA portal, (2) workflow running through the portal and (3) the Web services launched by the workflow on remote hosts. So when the signal comes on the portal to stop the workflow, then the services must stop executing and workflow process should come to stand still before the workflow is killed. This way all the threads created by the workflow on portal will end without leaving anything hanging. To achieve this, the entity 2 mentioned above must listen to the previous entity 1, which issues a signal that entity 2 is listening. Entity 2 and 3 communicate, but entity 2 must pass a signal to the service to stop the job when it receives a message from entity 1. So entity 2 in the middle plays a dynamic role in listening to messages from entity upstream and communicating to entity downstream.

As depicted in Fig. 14.4, we start with the launch of workflow from the portal. Each workflow launch is associated with a unique taskId, which is created prior to the start of workflow and stays with the process on portal. We have a new a component, resource monitor, added to the workflow and it is looking for a resource that bears the name of taskId either on a file system or a URL. The resource monitor (RM) is the key element in entity 2 that is listening and communicating with the upstream and downstream entities respectively see Figs. 14.4 and 14.5. When the user (or administrator) signals intent to terminate the job, prior to the termination of workflow, the taskId resource is created that RM is constantly looking for. Upon finding it, RM resets a variable on the workflow such that the command check job status is changed to terminate. Hence, all the processes started by Web services upon the launch of a particular workflow, that is associated with a taskId, terminate. Subsequently, the main process sleeps for a brief period such all the threads end and followed by workflow process termination as illustrated in Fig. 14.5. This way, the entire lineage associated with a workflow or taskId finishes gracefully.

Unit Testing in Loosely Coupled SOA Environments

In a loosely coupled environment such as CAMERA, testing has been a challenge due to the interaction of many services with our applications. CAMERA has code

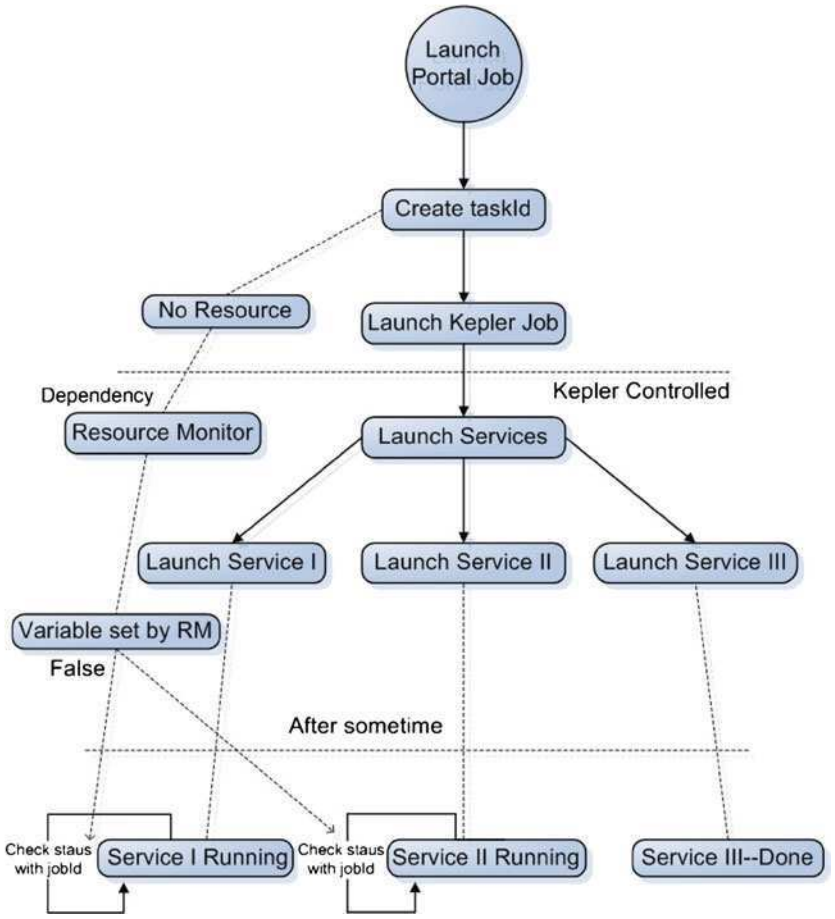


Fig. 14.4 Portal workflow job launch sequence

written in Java (www.java.sun.com), Perl (www.perl.com), and PHP (www.php.net) developed by several separate groups within the project adding additional complexity to testing. This section describes how testing works in the CAMERA system and the challenges encountered.

CAMERA testing starts at the unit test level, or the testing of individual methods or classes. For applications written in Java this is done using Junit (www.junit.org) with Ant (ant.apache.org) or Maven (maven.apache.org) build targets to call the tests. In Perl the module Test::More (http://search.cpan.org/~mschwern/Test-Simple-0.96/lib/Test/More.pm) is used with a Makefile target called to invoke the tests. Below are examples of unit tests in Java and Perl. In the first example below the unit test is verifying that a call to the **submitWorkflowJob** method with a null parameter results in an exception.

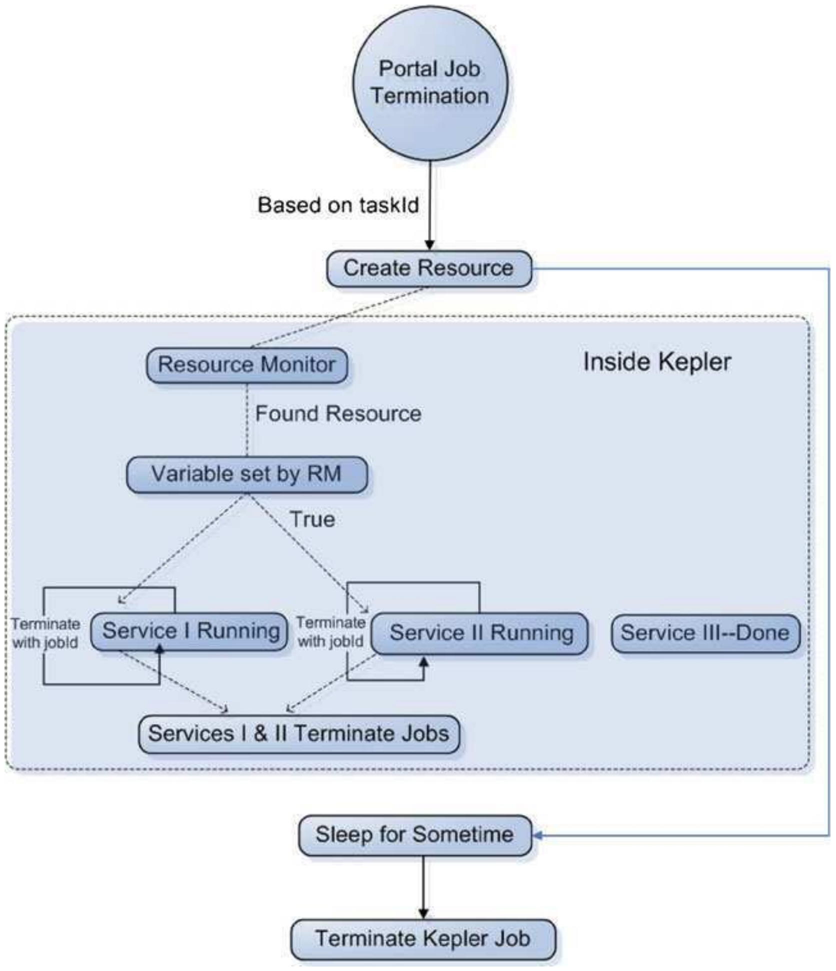


Fig. 14.5 Portal workflow termination steps

Example of a Junit test

```
@org.junit.Test
public void testSubmitWorkflowJobWithNullArg() throws Exception {
   try  {
        keplerSGEClient k = KeplerSGEClient.getInstance();

        //call submitWorkflowJob with null argument which should
throw exception
        k.submitWorkflowJob(null);

       //Fail test because we did not get the exception
      fail(''Expected IllegalArgumentException'');
   }
   Catch(IllegalArgumentException ex){
      //Caught exception and verify message is correct
      assertTrue(ex.getMessage().contains(``Null WorkflowTask''));
   }
}
```

The Perl tests below verify that get and set methods in the User class work correctly.

Example of a Perl unit test

```
#test empty constructor and get/set of *Login() methods
{
   #create user object
   my $user = WorkflowSandboxer::User->new();

   #verify user object was created
   ok(defined($user));

   #verify call to getLogin() returns undef on newly created User
object
   ok(!defined($user->getLogin());

   #set login for user to ``foo''
   $user->setLogin(``foo'');

   #verify getLogin returns expected value of ``foo''
   ok($user->getLogin() eq ``foo'');
}
```

While unit testing of code locally developed and deployed by CAMERA is relatively straightforward, a particular challenge is the high cost pertaining to setup of external services needed for unit testing some of the code base. A current solution employed by the CAMERA project is mock objects. A mock object is a "fake" implementation of a class used to simplify testing. Below is an example of a Java unit test making use of an SGE mock Session object created with EasyMock (www.easymock.corg).

In the example below a mock Distributed Resource Management Application API (DRMAA, www.drmaa.org) session has been created and the mock object is created to assist in testing getJobStatus method whose job is to get the status of a job from DRMAA and convert it to a human readable string. In the above case the mockSession object is set to expect a call to getJobProgramStatus and to return the integer 1. The unit test then verifies the method getJobStatus returns the correct value and the verify method checks that the call was actually made to the mock object.

Java Unit testing with EasyMock

```
@org.junit.Test
public void testGetExeStatusWithValidJob() throws Exception {
    //get instance of KeplerSGEClient
    KeplerSGEClient kepSGEClient = KeplerSGEClient.getInstance();

    //create mock of SGE Session
    Session mockSession = createMock(Session.class);

    //Tell mock Session to expect a call getProgramStatus
(‘‘193434’’) and to return 1
    expect(mockSession.getJobProgramStatus(‘‘193434’’)).
andReturn(1);
    //set the mock Session object in KeplerSGEClient
    kepSGEClient.setSession(mockSession);

    //tell mock object to respond to user method call defined
above
    replay(mockSession);

    //call getJobStatus on KeplerSGEClient with job id 193434
    //and check that the call returns Submitted
    assertTrue(k.getJobStatus(‘‘193434’’).equals(‘‘Submitted’’));

    //check that getProgramStatus was invoked on mock Session
    verify(mockSession);
}
```

Mock objects are great for enabling unit tests to be written that exercise code which call external services and systems, but from time to time the behavior of the mock object has not matched that of the service resulting in a failure of the system during the integration step.

The next level of testing employed at CAMERA is known as system tests, or tests that exercise aspects of the entire system. For systems and services this testing is done by including testing programs that are deployed along with the application to the various environments. These programs call the services and systems the same way a user would and include code to verify correct operation.

One example of this is with the Kepler workflow system tests, which are written in Java using Junit and are invoked by an ant build target in the workflow build source tree. Below is an example of invoking a system test on the Blastn workflow where the unit test is invoking the workflow the same way a user would if logged into the system.

Example of running system test on Blastn workflow via Ant

```
$ ant workflowtest --Dworkflowname=’Blastn’
workflowtest:
    [junit] Testsuite: net.calit2.camera.WorkflowTestBlastn
    [junit] Tests run: 4, Failures: 0, Errors: 0, Time elapsed:
189.018 sec
    [junit] Testcase: testDelimiterSetCorrectly took 0.397 sec
    [junit] Testcase: testBlastnDefaultParams took 69.887 sec
    [junit] Testcase: testBlastnWithCAMERARefDatasetAndAltValsFor
```

```
Params took 59.153 sec
    [junit] Testcase: testBlastnTestTooManyAlignmentsError took
59.284 sec

BUILD SUCCESSFUL
Total time: 3 minutes 11 seconds
```

While technically this isn't a total system test as the workflow being tested is local to the source tree, the services and processing are on the appropriate target environment (development, stage, or production).

The system tests have proven to be very valuable in checking system integrity and as an automated way to check new releases, but issues have arisen. One issue is these tests can take several hours to completely run. Another issue is they require access to the cluster which is quite busy and necessitates putting on hold real user jobs. It should also be noted these system tests do NOT test the user interface portion of the code base which is left to manual testing at time of release.

Fully testing SOA such as CAMERA that contains both internal and externally developed software is an ongoing challenge. While there will likely always be discrepancies between tests and reality, and impact of testing on production, the benefits have outweighed the costs and appear to have improved the quality of the system.

Automated Build and Deployment for SOA

Services and applications that comprise of the CAMERA system each have their own source tree and build systems and are owned by several different development groups within CAMERA. In addition, some of the applications predate the CAMERA group and already have preset configurations for setup and deployment. It is in this context that a deployment system needed to be developed.

Rather than attempt to force all these diverse projects into a single code base it was decided to leave everything where it was, but to setup a consistent way of configuration and deployment for each application and to create a master build and deploy project that could checkout, build, and deploy the entire software stack. This master project is known, for lack of a better name, as camera_build_and_deploy. Figure 14.6 summarizes what camera_build_and_deploy does.

The entire deployment system is based upon a few simple premises. The first premise is that each server to host one or more applications must be "provisioned" with proper system software and configuration to support the application. Second, each application must be able to consume a properties file that defines all configurable aspects of the application including where to deploy. Third, each application must be able to deploy itself via scp and ssh to its target host. This includes making calls to stop and start appropriate services.

The camera_build_and_deploy project contains logic in its build file to checkout all these applications along with a properties file that merely lists properties files for all the applications it is to deploy. Below is a exerpt from the camera_build_and_deploy properties file.

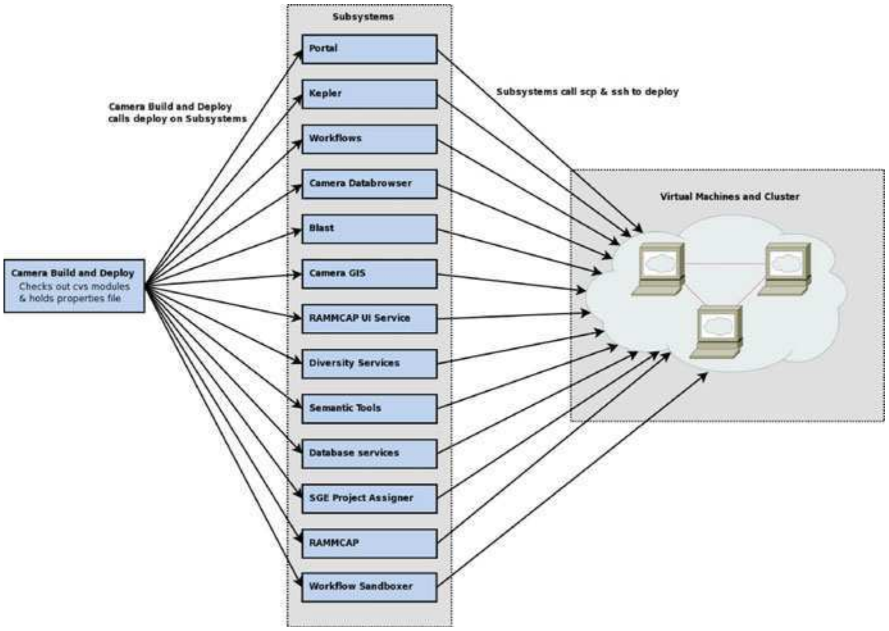
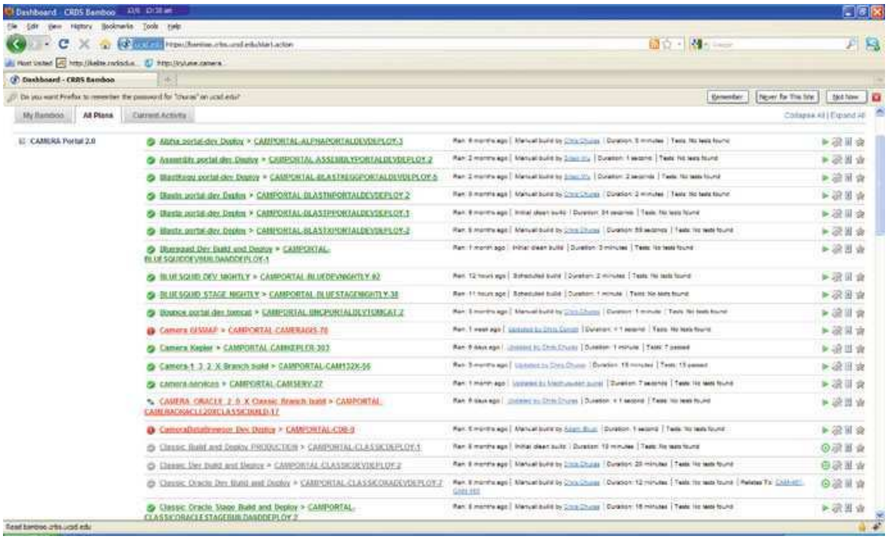


Fig. 14.6 Camera build and deploy system

Example of build_and_deploy properties file

```
# Path to portal properties file relative to cameraportal cvs
module portal.properties=portal-dev.properties

# Path to classic portal properties file relative to camera/
buildprocess

#in camera cvs module
classic.properties=config/portal-dev.properties

# Path to kepler properties file relative to base directory of
# workflows/camerakepler in cvs
kepler.properties=properties/portal-dev.properties

# cvsroot used by camera_build_and_deploy to do cvs operations
cvsroot=:ext:..@..:/CVS

# cvs tag to check out classic app
classic.cvs.tag=CAMERA_ORACLE_2_0_X
```

Below is an example of checking out camera_build_and_deploy as well as invoking the deploy target which would deploy the entire system.


Dashboard - CRBS Bamboo

CAMERA Portal 2.0

Alpha portal-dev Deploy > CAMPORTAL-ALPHAPORTALDEVDEPLOY-1
Assembly portal dev Deploy > CAMPORTAL-ASSEMBLYPORTALDEVDEPLOY-2
BlastRegq portal-dev Deploy > CAMPORTAL-BLASTREGQPORTALDEVDEPLOY-5
Blastn portal dev Deploy > CAMPORTAL-BLASTNPORTALDEVDEPLOY-2
Blastp portal-dev Deploy > CAMPORTAL-BLASTPPORTALDEVDEPLOY-1
Blastx portal-dev Deploy > CAMPORTAL-BLASTXPORTALDEVDEPLOY-2
Bluesquid Dev Build and Deploy > CAMPORTAL-BLUESQUIDDEVBUILDANDDEPLOY-1
BLUESQUID DEV NIGHTLY > CAMPORTAL-BLUEDEVNIGHTLY-82
BLUESQUID STAGE NIGHTLY > CAMPORTAL-BLUESTAGENIGHTLY-38
Bounce portal dev tomcat > CAMPORTAL-BNCPORTALDEVTOMCAT-2
Camera GSWAP > CAMPORTAL-CAMERAGIS-70
Camera Kepler > CAMPORTAL-CAMKEPLER-303
Camera-1.3.2.X Branch build > CAMPORTAL-CAM132X-56
camera-services > CAMPORTAL-CAMSERV-27
CAMERA_ORACLE_2.0.X Classic Branch build > CAMPORTAL-CAMERAORACLE20XCLASSICBUILD-17
CameraDataBrowser Dev Deploy > CAMPORTAL-CDB-9
Classic Build and Deploy PRODUCTION > CAMPORTAL-CLASSICDEPLOY-1
Classic Dev Build and Deploy > CAMPORTAL-CLASSICDEVDEPLOY-2
Classic Oracle Dev Build and Deploy > CAMPORTAL-CLASSICORADEVDEPLOY-2
Classic Oracle Stage Build and Deploy > CAMPORTAL-CLASSICORACLESTAGEBUILDANDDEPLOY-2

Fig. 14.7 Screenshot of bamboo Web interface showing build plans

Deploy call for the entire system

```
cvs co camera_build_and_deploy
cd camera_build_and_deploy
ant -Dproperties.file=properties/prod.properties deploy
```

Invoking camera_build_and_deploy above would invoke a command similar to the one below for every application the camera software stack. In the example below, the blast service and Web interface is built and deployed to environment defined in prod.properties file:

Deploy call for blast service and Web interface

```
cvs co camera
cd camera
ant -D properties.file=config/prod.properties -f build-all.xml
deploy-all
```

A final piece of automation is the use of Bamboo (www.atlassian.com/software/bamboo/) to manage invocation of the deployment and build targets. Bamboo also provides continuous integration by automatically calling builds after developers check in code. Figure 14.7 is a screenshot of the Web interface to Bamboo where developers and administrators, who have appropriate permissions, can run build targets.

Overall this approach has worked well, with the main issue being a lot of configuration when a property changes since it requires a developer to examine and modify values in multiple property files. Even this issue has been minor considering the number of applications that have to be deployed that make up the CAMERA software stack.

Discussion

SOA has become a de-facto standard in application development. Particularly in the case of scientific applications, it is increasingly rare for applications to be self-contained and running on single workstations. Rather it is more likely that core computational and data intensive components be performed on dedicated distributed resources connected by services.

REST has provided a central foundation that addresses CAMERA's unique SOA requirements. Because CAMERA allows community-based development of services and workflows, we must provide a specification that is simple yet does not limit functionality. Because REST is stateless with a well-defined CRUD interface, it was a natural fit. Moreover, by using REST, we create a natural separation of concerns in application-state management. With workflows managing state, we enable a full provenance record that is critical to scientific (and other) application environments.

With SOA, come unique challenges that must be addressed. To maintain a robust resource, we continue to refine our tests. Again, the simple CRUD interface is an asset. Knowing that the services have a well-defined scope, we can build automated systems to systematically test all service components. Likewise we continue to refine automated deployment mechanisms. The termination of work is also a challenge with SOA. We have found that the use of an "observer pattern" works well to make sure all services in the process are terminated.

Acknowledgements This work is supported by the Gordon and Betty Moore Foundation.

References

Deelman, E., Gannon, D., Shields, M. and Taylor, I. (2009) Workflow and e-science: an overview of workflow system features and capabilities. FGCS, 25, 528–540.

Ludaescher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger, E., Jones, M., Lee, E.A., Tao, J. and Zhao, Y. (2006) Scientific workflow management and the Kepler system. Concurrency Comput. Pract. Exp., 18, 1039–1065.

Sun, S., Chen, J., Li, W., Altintas, I., Lin, A., Peltier, S., Stocks, K., Allen, E.E., Ellisman, M., Grethe, J., and Wooley, J. (2010) Community cyberinfrastructure for advanced microbial ecology research and analysis: the CAMERA resource. Nucl. Acids Res., 1–6. doi: 10.1093/nar/gkq1102

Chapter 15
Practical REST in Data-centric Business Applications: The Case of Cofidis Hispania

Jordi Fernandez and Javier Rodriguez

Thank you very much to Esther Vidal, Jordi Albert, Albert Espelt and Oriol Garcia for their commitment to this project. Great job!

Abstract This chapter describes the migration of the IT environment in an important financial institution, from a mainframe-centric to a Web-centric environment in which the REST architectural style had a key role in the reference architecture that supported the new software development projects. We will describe how the restrictions imposed by the REST architectural style addressed the most critical constraints as well as some other challenges by means of a real-world, three-year project that is still ongoing at the time of writing. In particular, we will detail how each of the restrictions of the REST architectural style has contributed to address different software architecture requirements, both functional and non-functional, and how they have been materialized in the Java platform. We will detail advantages and compromises, strengths and weaknesses, and areas with the most interesting challenges.

Background, Constraints, and Challenges

In a story that is too common in the financial sector, Cofidis Hispania has relied for decades in its legacy mainframe systems for day-to-day business. In this market, applications tend to be heavily data-centric, relying in central databases to store information about customers, contracts, accounts payable, and interaction with other financial institutions. All this data must be readily available to maintain a growing set of critical indicators about credit, debt, risk and profitability.

In 2005, Cofidis was running most of its operation through an application suite residing in the mainframe and accessed through terminal emulators from commodity PCs from all areas of the company, including the Call Center – the heart of the operation and the customer-facing side of the business. The legacy application suite

J. Fernandez (✉)
Esilog Consulting, S.L., Aribau 112, Barcelona, Spain
e-mail: jordi.fernandez@esilog.com

was responsive and reliable, but imposed a steep learning curve on all new hires, requiring extensive training and with a low margin for human error. Moreover, the organization was facing a drying supply of skilled COBOL programmers, which translated in long development times even for the tiniest maintenance changes, making it harder to create new software modules to support new products and maintain the lead in a growing market. As consultants, the challenge was to provide Cofidis with an agile development environment that allowed the company to leverage its IT resources for competitive advantage.

The suggested approach was to create an interface layer to phase out the legacy systems in favor of the Java Enterprise platform, so new modules could be built in a matter of weeks instead of months, using skills readily available from a rich pool of consulting firms with a java-centric software development practice.

Building a web-based corporate software platform is a challenge that needs to take into account the needs of different stakeholders besides those of the end users: those of the executives, project leaders, system administrators and software development teams just to name a few.

The stakeholders imposed a set of constraints that created an interesting challenge from the software architecture standpoint. Some of the most interesting were:

- Adopting a strict, ACID[1] – compliant transaction approach at all levels as a definitive business requirement. In a data-centric organization it is essential to maintain a reliable database at all times.
- It was necessary to take into account the integration points with diverse external systems – the mainframe during the transition period, all PBX[2] and CTI[3] infrastructure, ERP[4] and accounting systems, and so on.
- Build Web-based applications with a level of responsiveness that compares favorably with the (then) current user experience, which was based on using a terminal emulator to establish telnet sessions to the mainframe.
- Make the best effort to mitigate the impact of the transition from an environment that required mastering a single programming language (COBOL) to another that required knowledge of several languages (Java, Javascript) and markup languages (XHTML, XML).
- Define a common software architecture that provides guidelines to the efforts of software developers so that several software providers could participate in the

[1]Atomicity, consistency, isolation, durability (ACID) is a set of properties that guarantee database transactions are processed reliably.

[2]A private branch exchange (PBX) is a telephone exchange that serves a particular business or office, as opposed to one that a common carrier or telephone company operates for many businesses or for the general public.

[3]Computer telephony integration, also called computer–telephone integration or CTI, is a technology that allows interactions on a telephone and a computer to be integrated or co-ordinated.

[4]An Enterprise Resource Planning (ERP) system is an integrated computer-based application used to manage internal and external resources, including tangible assets, financial resources, materials, and human resources.

building of the platform while keeping coherence among developments from the architecture standpoint.
- Provide a reasonable migration path for legacy systems, so live with new developments while maintaining consistency of data.
- Achieve maximum business logic code reuse among web, batch or rich client applications – batch processing is a staple in most financial institutions.
- Switching client tier technology should be supported by the architecture. Aim for a web-based client tier in the first phase, but different client tier presentation technologies should be explored, including RIA and rich desktop client (such as Eclipse RCP).
- Call-center employees usually work simultaneously with two or more customers. A single employee will use more than one browser instance at any given time.
- Scalability is a principal concern. There is high concurrency and high peak loads. Besides, the business is growing rapidly and it is expected that this same software platform would serve other countries.

Besides the explicit restrictions manifested by the stakeholders, we need to keep in mind other highly relevant implicit restrictions. The most obvious are those related with moving from a local software development context – COBOL programs running in a mainframe – to a distributed programming environment such as an Intranet built using the same technologies used for the World Wide Web. And moving from a local environment to a distributed programming environment presents a particular set of challenges as identified by Waldo et al. (1994): latency, a different model of memory access, and issues of concurrency and partial failures.

Reference Architecture and the Role of the REST Architectural Style

One of the ideas that we try to inseminate into our software development teams is that – in the great majority of cases – we won't be the first entity to encounter a given software development challenge, so it is always convenient to check out if there is an existing solution to the particular problem we are facing. The same concept applies from a software architecture standpoint: we saw clearly that in order to guarantee the coherence among the multiple development modules, all of them should exhibit certain architectural qualities that captured the aforementioned requirements. It is very likely that other persons have faced this same challenge, so we should take advantage of all existing knowledge. We have at our disposal many forms of reutilization at the architecture level, including Domain Specific Software Architectures, Architectural Patterns, Architectural Styles and Design Patterns.

The reader may be familiar with the concept of Design Pattern as presented by Gamma et al. (1994). Design patterns offer excellent design solutions in the context of object-oriented programming, but this kind of solutions do not apply to systems design at the enterprise level. At this scale we will find Domain-Specific Software

Architectures (DSSAs), which typically disclose deep knowledge acquired through experience about how to structure applications in a given domain. Between Design Patterns and DSSAs we find Architectural Styles and Architectural Patterns. An Architectural Style is a named collection of architectural design decisions that are applicable in a given development context, constrain architectural design decisions that are specific to a particular system within that context and elicit beneficial qualities in each resulting system (of course, a suitable example is the REST style Fielding (2000)). An Architectural Pattern is very similar to a style: It is a collection of architectural design decisions that are applicable to a recurring design problem, parameterized to account for different software development contexts in which that problem appears. A typical example in distributed systems is the three tier system pattern: client tier, business logic and back-end. In fact, Architectural Styles and Architectural Patterns are very similar. Taylor et al. (2010) identifies three important differences:

Scope Architectural Patterns are of narrower scope, targeted to a design problem ("presentation logic must be separated from business logic") while Architectural Styles are of broader scope, applying to a development contexts such as "highly distributed systems".

Abstraction Architectural Styles are more abstract than Architectural Patterns. The former constrain the architectural design decisions about a system but are too abstract too offer a concrete system design while the latter are parameterized architectural fragments that can be thought as concrete pieces of a design.

Relationship A single Architectural Pattern could be applied to systems designed according to the guidelines of multiple Architectural Styles, and a system design according to the rules of a single Architectural Style may involve the use of multiple Architectural Patterns.

The proposed solution included the specification of a reference architecture, that is, a set of principal design decisions that are simultaneously applicable to multiple related systems, typically within an application domain, with explicitly defined points of variation. Having a reference architecture would allow us to adopt an strategy to ease building product families with high similarity in their factual architectures, that is, regarding the main design decisions. Moreover, this would allow a generative strategy in regards to implementation: an important part of the source code of the application could be generated automatically by a software tool. Thus, from an architectural point of view a good portion of the implementation would be of a very high quality from their inception, since all generated applications would exhibit consistently all principal design decisions.

The REST architectural style allowed us to take advantage of several design principles and their corresponding constraints in order to satisfy an important set of the challenges imposed by the project at the architecture level, so it was used in the reference architecture. It is not the aim of this chapter to describe the reference architecture, but it is to detail how the REST style contributed to it.

Challenges Addressed by REST Constraints

The REST architectural style consists in a set of constraints that, when applied to a software system, some beneficial qualities arise. This section will show the reader how every restriction of the REST architectural style addressed some of the most important challenges of this project.

Client–Server

The client–server constraint provides a clear separation between the initiators of communication (clients) and those who perform the required functions (server). This means that multiple and different clients can communicate with the server as long as the clients respect the interface offered by the server.[5] Clients will be able to evolve independently of servers.

A clear separation between clients and servers enable a diversity of clients. Initially, the client of the newborn platform would be a web browser, but the solution should allow the adoption of other client technologies if necessary. In fact, the interaction with the mainframe during the transition period practically involved a client that is not a web browser. Besides, having the business logic reside in the server permits to focus all transactional activity in a single place, as near as possible to the transaction-aware components such as the database, in order to minimize the effects of locci distribution.

Stateless

In our opinion, avoiding to maintain state in the server side is a crucial element for the success of any web-centric software development project. When we maintain state in the server, we open the door to a multitude of problems that will show up progressively through the project, either during development or testing in a best-case scenario, but the nastier bugs will linger in the dark, waiting to show up in a critical production phase. An entire legion of developers has been raised using (and abusing) the session object that practically every web development framework leaves within their reach. Using this object to store the conversational state of a web interaction is an error, plain and simple. This does not mean that there are no legitimate uses for the session object – caching data for a specific user is a perfectly honorable use of this resource.

[5]We will see in "Uniform Interface" how the Uniform Interface constraint plays a key role here.

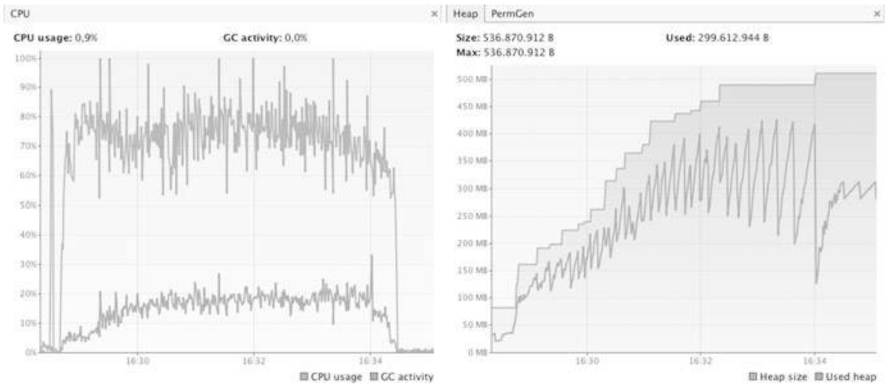


Fig. 15.1 CPU and Heap memory usage for a stateless RESTful application under 350 stressing request threads. After a transient period the memory usage stabilizes to around 250 Mbytes

The stateless constraint means that each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is, therefore kept, entirely on the client. In the moment that the server remembers the state of the session we are binding the client – a particular instance of a web browser – to that server. If the server crashes and the client is redirected to a different server, the request will fail because the state of that particular conversation is missing in the new server. Yes, we are aware of the existence of session clusters in JEE application servers, but why would anyone want to increment the complexity of a system when there is a simpler approach to the same means through stateless interactions?

Another typical problem related to keeping state in the server is when the user decides to open a new window in the same instance of the web browser. Both windows interact with the application in the context of the same session. This is a subtle situation that only complicates session management and is highly conductive to error.

Let us imagine for a moment that a rookie developer decides to store in the server-side session object the customer ID of the current customer. Do you remember the aforementioned requirement of letting a user working with several customers at once? This situation involves having several windows open at once, each presenting data for a different customer. Keeping the customer information in the session object makes it impossible to work reliably with several customers at once. But if we keep the state in the client – using a hidden field for the customer ID, for instance – the problem disappears.

There is a noticeable effect in terms of scalability for the stateless quality of the REST architectural style. Keeping state in the client allows the server to free its resources faster, since resource utilization is constrained to a single HTTP request.

In practice, having a stateless application has brought benefits along several lines. First, the usage of heap memory in the server tends to stabilize over time for a given number of users. Figure 15.1 shows the CPU and memory usage for a 350 request threads stressing the server without waiting intervals (we are then simulating much more than 350 users). After a short transient period the memory usage settles to around 250 MB in heap space.[6] Second, by avoiding the usage of the session and application scopes we gain in development simplicity, as there is no need to synchronize concurrent access to shared resources. Third, we avoid all problems associated to a user that initiates a session in a different browser but using the same credentials.

Cache

Making the response times of a web application comparable to those of a telnet session is a challenge that just has to take advantage of the concept of caching as much as possible. An efficient use of caching can minimize data transfer for certain requests and even eliminate the necessity for some of those requests in the first place.

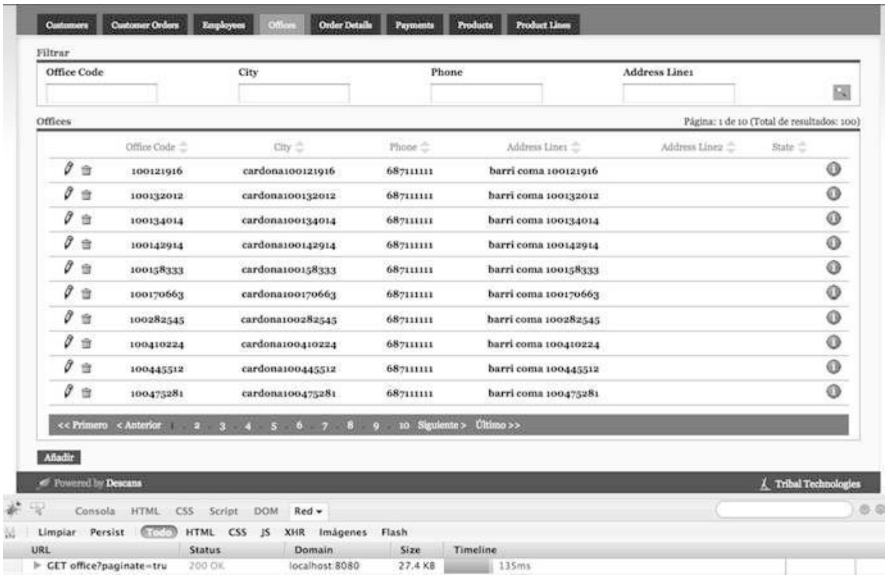
The HTTP protocol offers several mechanisms for cache usage: expiration, validation, and a combination of both (Tomayko 2010).

The expiration model allows the server to indicate in its response that it is valid only for a certain time frame (e.g. 120 s) or until certain date and time (e.g. December 4th, 2011, at noon). The client can avoid requesting again that resource as long as it has not reached its expiration date.

In the validation model, the server provides a code (an Etag) associated to the representation of a given resource (some kind of "hash code"). When the client asks for a new representation of that resource it will send the code along with request. The server will be able to determine if the representation has changed in the intervening time frame. If it has not changed, it will use an HTTP 304 response code to signal the client that the copy in its cache is still valid.

Cache usage, both in its expiration and its validation models, allowed the construction of user interfaces with an optimal perception of latency from the user viewpoint. In order to take the most advantage of the cache facilities in HTTP, we made a few modification to the Struts2 REST plugin so that in practice every response is qualified with HTTP cache headers. All static resources follow a default 24-h expiration model, and dynamic resources implement the ETag validation model. A 304 response code is transmitted whenever the requested resource is unchanged as determined by comparing of the client-supplied and the server-computed Etag, and furthermore the corresponding view is kept unprocessed. In the case of a XHTML representation we avoid the processing of a JSP with a significant performance gain. In typical use case the processing dropped from 150–300 ms to 10–20 ms, as shown in Figs. 15.2 and 15.3.

[6]Not maintaining server-side session state is a key factor in the low memory usage footprint, but the intense use of singleton objects (via the Spring Framework) is of great help here as well.


Customers | Customer Orders | Employees | Offices | Order Details | Payments | Products | Product Lines

Filtrar

Office Code | City | Phone | Address Line1

Offices — Página: 1 de 10 (Total de resultados: 100)

Office Code	City	Phone	Address Line1	Address Line2	State
100121916	cardona100121916	687111111	barri coma 100121916		
100132012	cardona100132012	687111111	barri coma 100132012		
100134014	cardona100134014	687111111	barri coma 100134014		
100142914	cardona100142914	687111111	barri coma 100142914		
100158333	cardona100158333	687111111	barri coma 100158333		
100170663	cardona100170663	687111111	barri coma 100170663		
100282545	cardona100282545	687111111	barri coma 100282545		
100410224	cardona100410224	687111111	barri coma 100410224		
100445512	cardona100445512	687111111	barri coma 100445512		
100475281	cardona100475281	687111111	barri coma 100475281		

<< Primero < Anterior 1 2 3 4 5 6 7 8 9 10 Siguiente > Último >>

Añadir

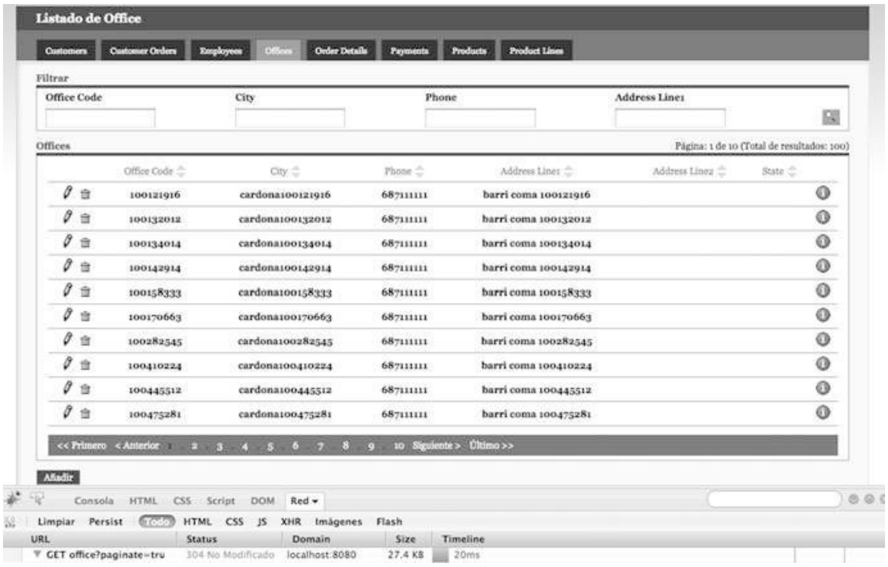
Powered by Descans — Tribal Technologies

Consola HTML CSS Script DOM Red

Limpiar Persist Todo HTML CSS JS XHR Imágenes Flash

```
URL                     Status    Domain          Size     Timeline
▶ GET office?paginate=tru  200 OK    localhost:8080  27.4 KB  135ms
```

Fig. 15.2 The first time we GET an XHTML representation it takes 135 ms for the browser to obtain a response as we can see in the Firebug console log


Listado de Office

Customers | Customer Orders | Employees | Offices | Order Details | Payments | Products | Product Lines

Filtrar

Office Code | City | Phone | Address Line1

Offices — Página: 1 de 10 (Total de resultados: 100)

Office Code	City	Phone	Address Line1	Address Line2	State
100121916	cardona100121916	687111111	barri coma 100121916		
100132012	cardona100132012	687111111	barri coma 100132012		
100134014	cardona100134014	687111111	barri coma 100134014		
100142914	cardona100142914	687111111	barri coma 100142914		
100158333	cardona100158333	687111111	barri coma 100158333		
100170663	cardona100170663	687111111	barri coma 100170663		
100282545	cardona100282545	687111111	barri coma 100282545		
100410224	cardona100410224	687111111	barri coma 100410224		
100445512	cardona100445512	687111111	barri coma 100445512		
100475281	cardona100475281	687111111	barri coma 100475281		

<< Primero < Anterior 1 2 3 4 5 6 7 8 9 10 Siguiente > Último >>

Añadir

Consola HTML CSS Script DOM Red

Limpiar Persist Todo HTML CSS JS XHR Imágenes Flash

```
URL                     Status            Domain          Size     Timeline
▼ GET office?paginate=tru  304 No Modificado  localhost:8080  27.4 KB  20ms
```

Fig. 15.3 The second time we GET an XHTML representation for the same resource with the appropriate cache headers the application "knows" that the representation has not changed since it was last requested, the JSP is not processed and a 304 response code is returned to the client (a web browser in this case). It now takes 20 ms for the browser to obtain a response (15% time as compared with 135 ms)

The reference architecture mandates that, whenever possible, all data obtained from the database or from a business service is cached by default. Of course, the web presentation layer takes advantage of this caching. But we must not forget that the batch processes will take advantage of this cache as well. An innocent-looking database query with a 5 ms cost that could be cached but is not can be overseen in a web application. But that same overhead during a hypothetical batch process that must process one million records in sequence will accumulate 5,000 s – roughly 1 h and 23 min – of unnecessary processing time.

One of the main challenges to tackle is the latency perceived by the end user. In many cases, when there is a migration from a host-based environment to a web-based application (a frequent occurrence in data-centered organizations) we have to face the fact that the users take for granted the response times of terminal-based applications, which are typically in the range of fractions of seconds. By leveraging the use of the cache mechanisms built into HTTP, an application can offer competitive response times and compare favorably against the terminal-based applications in terms of features – not only in terms of speed. In contrast, services must be carefully analyzed to determine what is cacheable and what is not. On the other hand, implementing caching as a cross-cutting concern involves the use of techniques that can be considered advanced, such as Aspect Oriented Programming. In short, caching is a first class citizen in the reference architecture.

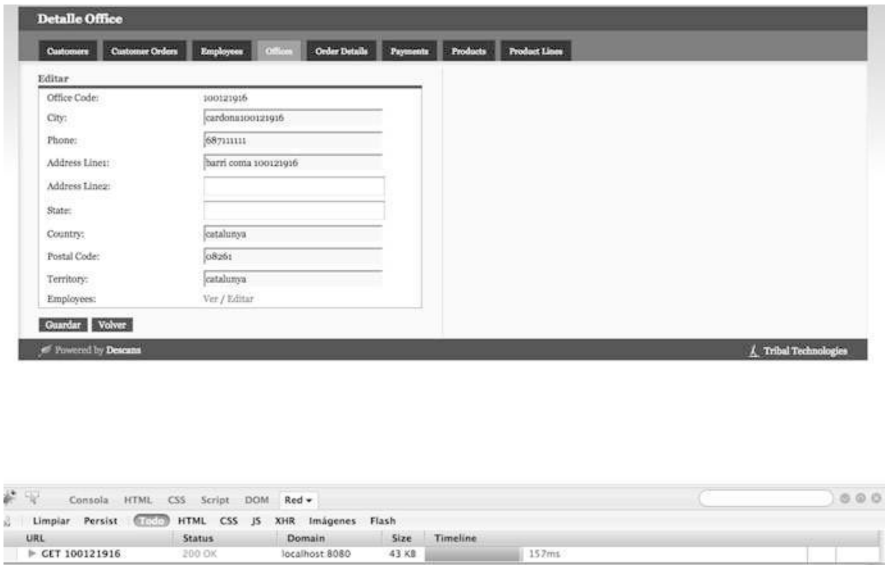
Uniform Interface

The uniform interface constraint of the REST architectural style contributes to a complete decoupling between clients and servers. This decoupling is achieved by the four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and hypermedia as the engine of application state.

Every relevant business concept should have its own resource identifier in the form of a URI. As we are in the context of a data-centered organization, the applications expose a unique URI for each entity inferred from the database, which can later be custom tailored. Every resource identifier corresponds to the fields in the primary key of the given entity. By adhering to this rule we obtain at least three benefits. (1) There is a uniform scheme to access any relevant business resource. (2) The business concept is separated from its concrete representation (XHTML, JSON, XML, etc.). (3) It is a mechanism to point to a concept through a permanent identifier (for instance, client and product are business concepts that have always existed, and that will most likely exist for a long time). Each domain model entity obtained from the database is exposed as a resource identified by its own URI (`/office, /employee, /product` , etc. We will see in "Tools and Frameworks" that this process has been automated with a code-generation tool.

```
[16-11-10 16:48:28] - 467818 INFO  org.apache.struts2.rest.RestActionInvocation  - Executed action [/office!index!xhtml!200] took 110 ms (execution: 22 ms, result: 88 ms)
[16-11-10 16:50:30] - 589926 INFO  org.apache.struts2.rest.RestActionInvocation  - Executed action [/office!index!xhtml!304] took 18 ms (execution: 18 ms, result: 0 ms)
```

Fig. 15.4 This log traces show how the cache constraint can dramatically reduce response times when GETting the same unchanged resource `/office` multiple times (`200 OK` HTTP response code with 110 ms response time for the first request but `304 Not Modified` response code with 18 ms response time on subsequent requests – no JSP processing)



Office Code:	100121916
City:	cardona100121916
Phone:	687111111
Address Line1:	barri coma 100121916
Address Line2:	
State:	
Country:	catalunya
Postal Code:	08261
Territory:	catalunya
Employees:	Ver / Editar

URL	Status	Domain	Size	Timeline
GET 100121916	200 OK	localhost:8080	43 KB	157ms

Fig. 15.5 An XTHML representation processing time for the `/office/100121916` resource

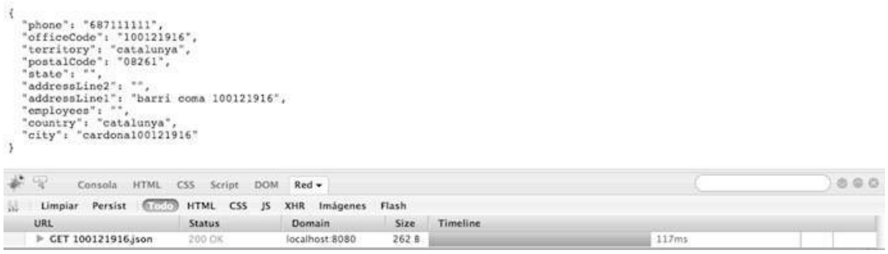
Resources in the reference architecture can be manipulated exclusively through a subset of the HTTP uniform interface: GET, POST, PUT, and DELETE. By using a uniform interface the services are completely and effectively decoupled from their clients, allowing independent evolution paths for both servers and clients.

Resources offer XHTML, JSON, XML, YAML, and ATOM representations, so it is possible to interact with the resources from a myriad of clients, who specify their preferred representation through a file extension (.xhtml, .json, .xml, .yaml, or .atom).[7] The JSON representation has allowed an intensive usage of AJAX in the presentation layer of most applications (this is related to the code-on-demand constraint). There is a significant improvement in performance whenever a Java application avoids processing a JSP template, and leveraging text-based representations like JSON is a good strategy to delegate template processing to the client and to free precious resources in the server side (Figs. 15.4–15.6).

Each generated entity representation supporting hypermedia (XHTML and ATOM) offer hyperlinks to related entities. These hyperlinks are inferred by the

[7]Use of `Accept` header is a work in progress.

application from the Primary Key and Foreign Key relationships in the database. Thus, the application is navigable in any of its representations, improving client–server decoupling and enabling easy workflow management by a programmatic client (Listing 15.1).



```
{
  "phone": "687111111",
  "officeCode": "100121916",
  "territory": "catalunya",
  "postalCode": "08261",
  "state": "",
  "addressLine2": "",
  "addressLine1": "barri coma 100121916",
  "employees": "",
  "country": "catalunya",
  "city": "cardona100121916"
}
```

```
URL                   Status   Domain           Size    Timeline
▶ GET 100121916.json  200 OK   localhost:8080   262 B   117ms
```

Fig. 15.6 A JSON representation processing time for the `/office/100121916` resource

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <feed xmlns="http://www.w3.org/2005/Atom"
3       xmlns:dc="http://purl.org/dc/elements/1.1/">
4       <title>employee</title>
5       <id>/show-web<id>
6       <updated>2010-11-17T08:18:15Z</updated>
7       <dc:date>2010-11-17T08:18:15Z</dc:date>
8       <entry>
9           <title>employeeNumber</title>
10          <author><name /></author>
11          <summary type="text">1088</summary>
12      </entry>
13      <entry>
14          <title>office</title>
15          <link rel="alternate"
16              href="http://localhost:8080/show-web/office/6.atom
                    " />
17          <author><name /></author>
18          <summary type="text">6</summary>
19      </entry>
20      <entry>
21          <title>lastName</title>
22          <author><name /></author>
23          <summary type="text">Patterson</summary>
24      </entry>
25      <entry>
26          <title>firstName</title>
27          <author><name /></author>
28          <summary type="text">William</summary>
29      </entry>
```

Listing 15.1 This is the ATOM representation of an employee which is related with an office and with a set of customers. Those relationships are explicit in an ATOM representations thanks to its hypermedia capabilities in lines 16 and 55. This representation can be customize to build a custom Domain Application Protocol as stated in Parastatidis et al. (2010)

```
    <entry>
        <title>extension</title>
        <author><name /></author>
        <summary type="text">x4871</summary>
    </entry>
    <entry>
        <title>email</title>
        <author><name /></author>
        <summary type="text">
            wpatterson@classicmodelcars.com
        </summary>
    </entry>
    <entry>
        <title>reportSto</title>
        <author><name /></author>
        <summary type="text">1056</summary>
    </entry>
    <entry>
        <title>jobTitle</title>
        <author><name /></author>
        <summary type="text">Sales Manager (APAC)</summary>
    </entry>
    <entry>
        <title>customers</title>
        <link rel="alternate"
 href="http://localhost:8080/show-web/customer.atom?
employee.employeeNumber=1088" />
        <author><name /></author>
        <summary type="text">6</summary>
    </entry>
</feed>
```

Listing 15.1 (continued)

An application under the reference architecture will use HTTP response codes strictly, such as `200` OK, `201` Created, `304` Not Modified, `400` Bad Request, `404` Not Found, `409` Conflict, `500` Internal Server Error or `501` Not Implemented (Fielding et al. 1999). This is a vital feature to achieve HTTP based integration with legacy systems and third-party software. Moreover, it reinforces the concept of a uniform interface and provides a consistent behavior among presentation layers based in different technologies.

The uniform interface, Hypermedia as the Engine of Application State (HATEOAS) and the capability to serve multiple representations allow a high level of decoupling between client and server, so it is possible to pursue gradual migrations of the presentation layer (from a Web browser to a Rich Internet Application, for instance) without altering the behavior of the server. In the worst case scenario, it would be necessary to add a new representation, but that does not affect existing clients. Most legacy systems can have seamless integration with RESTful applications as long as they have the capability to perform HTTP requests, which nowadays can be done in practically any platform from mainframes to embedded systems.

The integration capabilities are inherent to a RESTful interface. The qualities discussed above (uniform interface, multiple representations and strict use of HTTP response codes) allowed the integration of legacy systems with relatively small effort. Moreover, the system resilience improves considerably thanks to the semantics of the HTTP response codes for specific error conditions (`400 Bad Request`, `409 Conflict` or `503 Service Unavailable`, for instance) used in combination with the idempotent quality of the GET, PUT and DELETE methods. This allows a better, safer handling of partial failures, with built-in recovery features.

With these qualities at hand, integrating with diverse external systems (even with those we cannot anticipate right now) or providing a reasonable migration path for legacy systems is an achievable target.

Layered System

The layered quality of REST has allowed the incorporation of intermediary network elements between clients and servers, improving the growth of the infrastructure in network environments above the LAN mark. Moreover, the proxy acts as a central request and response nexus, improving monitoring and growth-prevision tasks. At an experimental level, Squid has been used as a RESTful proxy server for RPC-based legacy systems, acting as a so-called "Enterprise Service Bus" with all the benefits of a consistent interface but with reduced complexity, a manageable codebase, simplified configuration and easy administration. Squid as an intermediary is one of the more powerful actors in this kind of deployment; not only for caching/acceleration, but also load balancing, routing, and other services which comes at hand for one of our requisites: it is expected that this same software platform would serve other countries

RESTful-based Integration

Initially, integration with the legacy systems was achieved through the product supplied by the mainframe systems vendor, which at first was based in COM and later acquired support for SOAP. This enabled a consistent, vendor-supported coexistence between JEE applications and legacy mainframe code written in COBOL. But in the other direction, the path was not that crystal clear. In the first instance, even though there is a vendor-supported JVM for the legacy platform, its memory and execution profile made its cost prohibitive for many tasks. And the invocation path to access SOAP-based web services from COBOL legacy code is convoluted at best.

Keeping this in mind, the first benefits of a REST-enabled architecture were evident when the necessity arose to invoke business logic residing in the JEE platform from COBOL legacy programs. Instead of relying on a complex solution

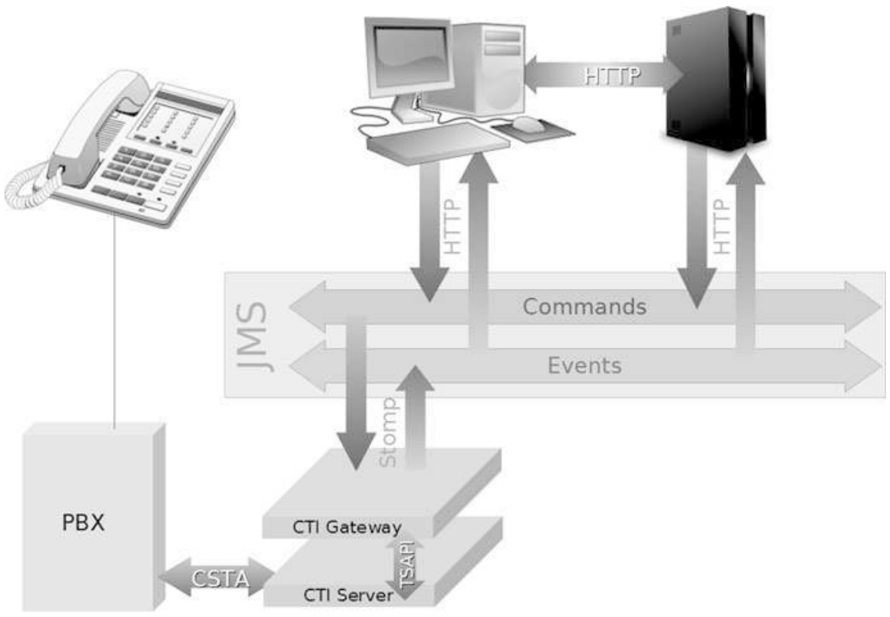


Fig. 15.7 This figure depicts how the different elements of the architecture interact regarding the CTI integration. The middleware take commands from a JMS queue and report CTI events back to a JMS topic in an Apache ActiveMQ server. Communication with the queues is exposed through a RESTful HTTP connector, allowing a web-based application to interact with the PBX for most CTI operations

based in Java or in a SOAP stack accessible from C and painfully integrated with COBOL, we wrote a tiny library for RESTful invocations using libCURL – a readily available HTTP access library – implemented a thin wrapper for invocation from COBOL programs in order to handle GET and POST requests through a consistent interface and take care of the finer points of HTTP request handling. In a matter of days, the mainframe was consuming RESTful web services in the Java platform.

A different challenge arose to interact with the CTI platform, which exposes a proprietary interface with libraries available for Win32 and Linux, but clearly intended to follow an strict client–server approach, with a middleware layer acting as server and clients running in user PCs.

In this scenario, installing access libraries and heavy clients in user PCs negates many of the side benefits of a web-based application deployment, mainly that there is nothing to install besides the browser itself. And in a heavily decentralized architecture like that suggested by the vendor, other concerns like performance monitoring, access control, and security become progressively more expensive in terms of implementation, supervision and maintenance.

In this case, we leveraged the REST architectural style by implementing a server with a thin layer of logic over the vendor-supplied CTI library in order to let the middleware take commands from a JMS queue and report CTI events back to a JMS topic in an Apache ActiveMQ server (see Fig. 15.7). Then we exposed those queues and topics through an HTTP connector, allowing a web-based application to interact with the PBX for most CTI operations. The server keeps track of the finer details of data conversion and state transitions, and delivers events to client web browsers through the Bayeaux protocol. As a side benefit, any application with an HTTP access library and the proper credentials can potentially become a client, issuing commands and subscribing to events. In theory, this will allow to move the burden of performance logging to a specialized analytics server, which will become a specialized application that will process events as any other client, filtering them according to a specific set of rules.

Tools and Frameworks

The code generation tool we have been using in this project is an in-house product that has been christened Alquimia.[8] In its most general form, it is a toolkit that leverages best-of-breed frameworks to generate RESTful Web applications for the Java platform. Alquimia was developed to support organizations that depend heavily on data, like financial entities or insurance companies. These organizations have in common two key characteristics: their business proposition relies deeply on data management and they have an ongoing need to integrate external systems. These characteristics translate into stable, long-lived data models and into multiple interfaces and integration points – be it to coordinate existing business processes with legacy systems or to synchronize information with ERP software, just to mention a couple of examples.

As a toolkit, we can think of Alquimia as a collection of components that for the sake of simplicity can be separated in two phases: code-generation on one hand, and run-time support libraries on the other. The code-generation logic resides entirely in a Maven 2 plug-in, and as such it can be integrated seamlessly in an existing build toolchain. The run-time support libraries take the form of several interceptors for Struts 2 and Spring, the latter acting as AOP logic. A key component is a series of improvements on the Struts 2 REST plug-in that have been contributed back into the upstream codebase.

The application generation process is as follows. Given a database model it only takes a few moments to configure the JDBC connection parameters and execute the proper Maven task in order to generate a RESTful Web application. This application will expose the database tables as domain entities, and effectively as REST resources.

Necessary training for application development using Alquimia is simplified considerably when compared against training for traditional Java/JEE-based development. The most common use cases observed in data centered applications are already implemented by Alquimia. Moreover, its RESTful interface offers an easy

[8] Alquimia will be opensourced in the near future. You can learn more at http://esilog.com/alquimia.

pattern for the developers to understand and follow. The amount of configuration is minimized as well, leveraging the convention-over-configuration approach. The target application is ready to add the necessary business logic and views.

Our experience with the generative approach offered by Alquimia has delivered existence proofs that such a technique improves productivity as well as maintains coherence among generated applications. We encourage others to follow this path when a domain specific architecture has been identified.

Conclusions and Future Work

The design principles of the REST architectural style is a fundamental piece of knowlegde ready to be reused in web based architectures. This chapter describes the key contribution it had in the reference architecture and the overall solution. We tried to show in this chapter how the abstract principles of REST applied to a real-world scenario successfully.

HATEOAS is a core tenet of the REST architectural style. If we can extend the HATEOAS capabilities of the hypermedia-enabled representations (e.g XHTML and ATOM) decoupling of clients and servers will improve. One of our objectives in the near future is to add the notion of Domain Application Protocol as stated in Parastatidis et al. (2010) in order to take full advantage of Hypermedia as the Engine of Application State. This way, domain specific business protocols will be exposed coherently by a family of applications.

We believe that in the near future we will be able to share our experience in this project through a Domain Specific Software Architecture document. We know that the same environmental constraints described in this chapter exist in many similar organizations. Through the DSSA document, future applications in this domain would be able to use this experience and influence their architectures. Most likely, new architectures would need only minor changes and architects would be able to focus in those changes, leveraging accumulated experience and fostering reutilization in the context of software architecture. Reutilization should not be constrained to architecture, since it can be extended to parts of the implementation and to the tools forged to support the creation of applications that implement the reference architecture. Actually, according to Taylor et al. (2010), a useful definition of a DSSA is the combination of (1) a reference architecture for the application domain, (2) a software component library for that architecture, and (3) a mechanism to choose and configure components that materialize an instance of the reference architecture. Through this progression we could contribute to stimulate the aforementioned concept of software production lines, which brings along an economic model with real knowledge and cash value for software development, demonstrating the relevancy of architecture-centered software development in general, and the benefits of the REST architectural style in particular.

References

Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. Phd Thesis, University of California, Irvine (2000)

Waldo, J., Wyant, G., Wollrath, A., Kendall, S.: A Note on Distributed Computing. Sun Microsystems Laboratories, Inc. (1994)

Taylor, R.N., Medvidovic, N., Dashofy, E.M.: Software Architecture. Foundations, Theory, and Practice. Wiley, NY, USA (2010)

Gamma, E., Johnson, R., Helm, R., Vlissides, J.M., Booch, G.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA (Wokingham, UK) (1994)

Parastatidis, S., Webber, J., Silveira, G., Robinson, I.S.: The Role of Hypermedia in Distributed System Development. WS-REST 2010 (2010)

Fielding, R.T., Gettys, J., Mogul, J., Frystyk, H., Masiner, L., Leach, P., Berners-Lee, T.: RFC 2616. Hypertext Transfer Protocol – HTTP/1.1 (1999) http://www.ietf.org/rfc/rfc2616.txt. Accessed October 2010

Tomayko, R.: Things Caches Do http://tomayko.com/writings/things-caches-do. Accessed October 2010

Part V
REST and Pervasive Computing

Chapter 16
RESTifying Real-World Systems: A Practical Case Study in RFID

Dominique Guinard, Mathias Mueller, and Vlad Trifa

Abstract As networked sensors become increasingly connected to the Internet, Radio Frequency Identification (RFID) or barcode-tagged objects are likely to follow the same trend. The EPC Network is a set of standards to build a global network for such electronically tagged goods and objects. Amongst these standards, the Electronic Product Code Information Service (EPCIS) specifies interfaces to capture and query RFID events from external applications. The query interface, implemented via SOAP-based Web services, enables business applications to consume and share data beyond companies borders and forms a global network of independent EPCIS instances. However, the interface limits the application space to the rather powerful platforms which understand WS-* Web services. In this chapter, we introduce tools and patterns for Web-enabling real-world information systems advertising WS-* interfaces. We describe our approach to seamlessly integrate RFID information systems into the Web by designing a RESTful (Representational State Transfer) architecture for the EPCIS. In our solution, each query, tagged object, location or RFID reader gets a unique URL that can be linked to, exchanged in emails, browsed for, bookmarked, etc. Additionally, this enables Web languages such as HTML and JavaScript to directly use RFID data to fast-prototype light-weight applications such as mobile applications or Web mashups. We illustrate these benefits by describing a JavaScript mashup platform that integrates with several services on the Web (e.g., Twitter, Wikipedia, etc.) with RFID data to allow managers along the supply chain and customers to get comprehensive data about their products.

D. Guinard (✉)
Institute for Pervasive Computing, ETH Zurich, Switzerland
e-mail: dguinard@guinard.org

Introduction

The EPC Network is composed of several standards addressing issues ranging from the Radio Frequency Identification (RFID) tags themselves (EPC standard) to readers infrastructure and the reading middleware (Floerkemeier et al. 2007). These standards define how to encode, read, and aggregate data about tagged objects throughout the whole supply chain. Furthermore, to be able to query and use recorded RFID data (i.e., traces), the EPCIS standard (Electronic Product Code Information Services) acts as a global track and trace sharing infrastructure with several, potentially interconnected, EPCIS servers distributed around the world. The EPCIS provides a simple and lightweight HTTP interface for recording EPC events. A different approach is taken to querying for these traces by other applications because the EPCIS specifies a standardized WS-* (i.e., SOAP, WSDL, etc.) interface. The WS-* integration architecture has been successfully used to combine business applications (Pautasso and Wilde 2009; Pautasso et al. 2008). For example, it can be used to integrate EPCIS data about the status of a shipment with an *Enterprise Resource Planning (ERP)* application.

However, WS-* applications are complex systems with a high entry barrier as it requires developer expertise in the domain. Hence, they are not optimal for more lightweight and ad-hoc application scenarios (Pautasso and Wilde 2009). Furthermore, the WS-* protocols are known to be rather verbose. Moreover, they do not fully meet the requirements of resource-constrained devices such as mobile phones and wireless sensor/actuator networks often not providing WS-* server or even client stacks (Yazar and Dunkels 2009; Luckenbach et al. 2005). As a consequence, these shortcomings limit the type of applications built on top of EPCIS servers to rather heavyweight business applications fully supporting the WS-* protocols. This is unfortunate since track and trace applications are also relevant beyond the desktop. As an example, providing out-of-the-box mobile access to this data might be beneficial for many users, in particular workers in storage rooms, transporters, etc. Similarly, providing direct access to RFID traces to sensor and actuator networks could enable those to react to RFID events. Finally, allowing lightweight Web applications (e.g., HTML, JavaScript, PHP, etc.) to directly access this data would enable the vast community of Web developers to create innovative applications using RFID traces.

In this chapter, we illustrate how a RESTful Application Programming Interface (API) for the EPCIS opens new design possibilities for RFID applications. First, it **lowers the entry barrier** for developers and fosters rapid prototyping. Second, it enables **direct access** to RFID data without any additional software other than the EPCIS itself. Direct access to EPC events allows to read, test, bookmarked, exchange, share RFID-related data from any Web browser, a tool ubiquitously available and understood by a vast number of people (Kindberg et al. 2002). Finally, it enables a more **lightweight** access to the data. This is particularly desirable for applications that need to access EPCIS data from **resource-constrained devices** such as mobile phones or sensor nodes. REST is known to be more light-weight (Yazar and Dunkels 2009) than WS-* services and many resource-constrained devices are REST-ready through simple HTTP client libraries or higher-level REST client libraries.

The chapter is structured as a "cookbook" each section begins with some theoretical background (recipe) and is then applied (cooked) to the implementation of the RESTful EPCIS. We start by briefly presenting the REST constraints. We then propose two implementation patterns and describe tools that can greatly speed up the development process of a RESTful enterprise system. Finally, we illustrate how REST fosters the "mashability" of real-world information systems with the EPC Mashup Dashboard. This Web mashup platform allows the exploration of EPC-related data and gathering of timely information about tagged objects from various Web services such as Twitter, Wikipedia, or Google Maps. Product or supply chain managers can use this tool as a business intelligence platform to better understand and visualize the entire supply chain. Likewise, customers can better understand and visualize where different products come from, what other people think about them, and so on.

Before looking at the "RESTification" process, we briefly introduce the EPC Network and summarize the basic concepts behind RESTful Web Services.

An Introduction to the EPC Global Network

As illustrated on Fig. 16.1, the EPC Network[1] is a set of standards established by industrial key players towards a uniform platform for tracking and discovering RFID tagged objects and goods. Fifteen standards are currently composing the EPC Network and addressing every step required from encoding data on RFID tags to reading them and sharing their traces. We will focus on two of them as those are the most relevant in the context of this paper.

The first standard is the EPC Tag Data Standard (TDS). It defines what an EPC number is and how it is encoded on the tags themselves as shown on the product box of Fig. 16.1. An EPC is a world wide unique number. Rather than identifying a product class, like most barcode standards do, it can be used to identify the instance of a product. The TDS specifies eight encoding schemes for EPC tags. They basically contain three types of information: the manufacturer, the product class and a serial number. As an example in the tag (represented in its URI form): urn:epc:id:gid:2808.64085.88828, `2808` is the manufacturer ID, `64085` represents the type of product and `88828` an instance of the product.

One of the goals of the EPC Network is to allow sharing observed EPC traces. Thus, the network specifies a standardized server-side EPCIS, in charge of managing and offering access to traces of EPCs events. Whenever a tag is read it

[1] http://epcglobalinc.org/standards/architecture.

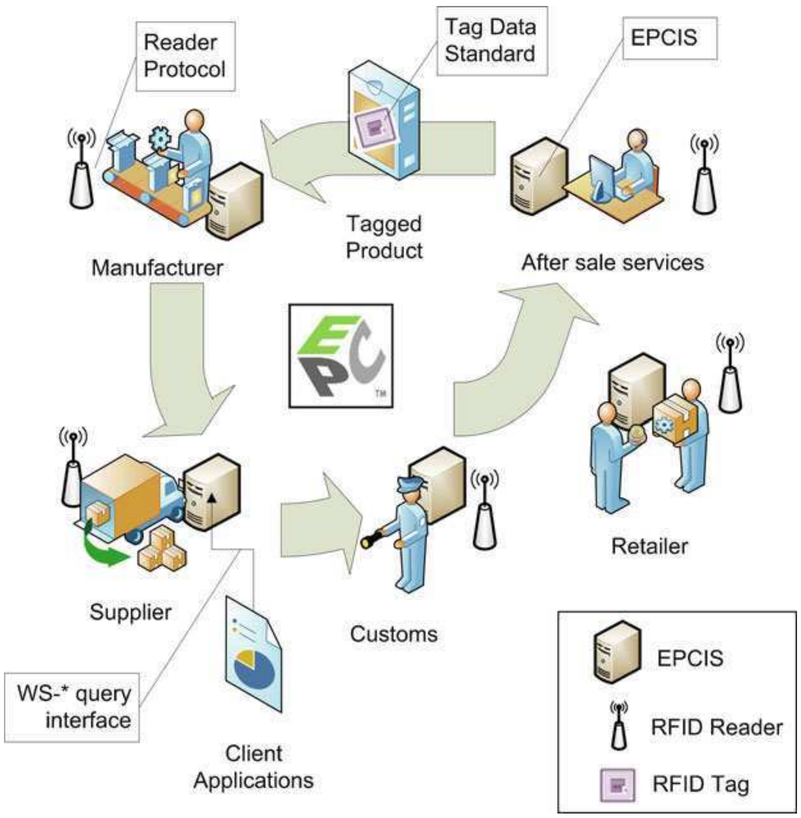


Fig. 16.1 Simplified view of the EPC Network and some of its main standards

goes through a filtering process and is eventually stored in an EPCIS together with contextual data. In particular, these data deliver information about:

- The “what”: what tagged products (EPCs) were read.
- The “when”: at what time the products were read.
- The “where”: where the products were read, in terms of Business Location (e.g., “Floor B”).
- The “who”: what readers (Read Point) recorded this trace.
- The “which”: what was the business context (Business Step) recording the trace (e.g., “Shipping”).

The goal of the EPCIS is to store these data to allow creating a global network where participants can gain a shared view of these EPC traces. As such, the EPCIS deals with historical data, allowing, for example, participants in a supply chain to share the business data produced by their EPC-tagged objects.

Technically speaking, a standard EPCIS is an application that offers three core features to client applications:

1. First, it offers a way to capture, i.e., persist, EPC events.
2. Second, it offers an interface to query for EPC events.
3. Third, it allows to subscribe to queries so that client applications can be informed whenever the result of a query changes.

There exist several concrete implementations of EPCISs on the market. Most of them are delivered by big software vendors such as IBM or SAP. However, the Fosstrak (Floerkemeier et al. 2007) project offers a comprehensive, Java-based, open-source implementation of the EPCIS standard.

The great potential of the EPC network for researchers in the ubiquitous computing field has led to a number of initiatives trying to make it more accessible and open for prototyping than it currently is. Floerkemeier et al. (2007) initiated the Fosstrak project, which is to date the most comprehensive open-source implementation of the EPC standards. The Fosstrak EPCIS is an open-source implementation of a fully-featured EPCIS. This project is suitable for prototyping (Floerkemeier et al. 2007) but it implements the standard WS-* interface which closes the EPCIS to a number of interesting use cases such as direct use from simple Web languages or usage on resource constrained devices.

To overcome these limitations, researchers started to create translation proxies between the EPCIS and their applications. Guinard et al. (2008) present an implementation of such a proxy. The "Mobile IoT Toolkit" offers a Java servlet based solution that allows to request some EPCIS data using URLs which are then translated by a proxy into WS-* calls. This solution is a step towards our goal as it enables resource-constrained clients such as mobile phones to access some data without the need for using WS-* libraries. Nevertheless, the proxy is directly built on the core of Fosstrak and thus does not offer a generic solution for all EPCIS compliant system. Furthermore, the protocol used in this implementation as well as the data format is proprietary which requires developers to learn it first.

In the "REST Binding" project,[2] a translation proxy is implemented, similarly to Guinard et al. (2008) it proposes using URLs for accessing the EPCIS data but these data are provided using the XML format specified in the standard. While this is an important improvement, the proposed protocol does not respect the REST principles but implements what experts sometimes call a REST-RPC style (Richardson and Ruby 2007). As we will explain in the next section, the connectedness and uniform interface properties do not held. Thus, an EPCIS using this interface is not truly integrated to the Web (Pautasso et al. 2008; Richardson and Ruby 2007). To better understand this, let us summarize some of the core notions of RESTful Web Services.

[2]http://autoidlabs.mit.edu/CS/content/OpenSource.aspx.

RESTful Information Systems

REST is an architectural style, which means that it is not a specific set of technologies. For this paper, we focus on the specific technologies that implement the Web as a RESTful system, and we propose how these can be applied to the Web of Things. The central idea of REST revolves around the notion of resource as *any component of an application that needs to be used or addressed*. Resources can include physical objects (e.g., a temperature sensors, an RFID tagged object, etc.) abstract concepts such as collections of objects, but also dynamic and transient concepts such as server-side state or transactions. REST can be described in five constraints:

- *Resource Identification*: the Web relies on *Uniform Resource Identifiers (URI)* to identify resources, thus links to resources can be established using a well-known identification scheme.
- *Connectedness: (also known as: Hypermedia Driving Application State)* Clients of RESTful services are supposed to follow links they find in resources to interact with services. This allows clients to "explore" a service without the need for dedicated discovery formats, and it allows clients to use standardized identifiers and a well-defined media type discovery process for their exploration of services. This constraint must be backed by resource representations, having well-defined ways in which they expose links that can be followed.
- *Uniform Interface*: Resources should be available through a uniform interface with well-defined interaction semantics, as is *Hypertext Transfer Protocol (HTTP)*. HTTP has a very small set of methods `GET`, `PUT`, `POST`, and `DELETE` with different semantics (*safe*, *idempotent*, and others), which allows interactions to be effectively optimized.
- *Self-Describing Messages*: Agreed-upon resource representation formats make it much easier for a decentralized system of clients and servers to interact without the need for individual negotiations. On the Web, media type support in HTTP and the *Hypertext Markup Language (HTML)* allow peers to cooperate without individual agreements. For machine-oriented services, media types such as the *Extensible Markup Language (XML)* and *JavaScript Object Notation (JSON)* have gained widespread support across services and client platforms. JSON is a lightweight alternative to XML that is widely used in Web 2.0 applications and directly parsable into JavaScript objects.
- *Stateless Interactions*: This requires requests from clients to be self-contained, in the sense that all information to serve the request must be part of the request. HTTP implements this constraint because it has no concept beyond the request/response interaction pattern; there is no native concept of HTTP sessions or transactions.

The design goals of RESTful systems and their advantages for a decentralized and massive-scale service system align well the field of pervasive computing: millions to billions of available resources and loosely coupled clients, with potentially millions of concurrent interactions with one service provider. Based on these observations, we argue that RESTful architectures are the most effective solution for the global Web of Things (Guinard et al. 2010), composed of smart appliances, sensor nodes and tagged objects. Indeed these architectures scale better and are more robust than RPC-based architectures like WS-* services.

Case Study: RESTifying the EPC Information Service

As mentioned before, in the EPCIS standard, most features are accessible through a WS-* interface. To specify the architecture of the RESTful EPCIS we systematically took these WS-* features and applied the properties of a *Resource Oriented Architecture (ROA)* we summarized in the previous section.

Resource Identification and Connectedness

All the services of a Resource Oriented Architecture are modeled with resources which are components of an application worth being uniquely addressed and linked to. Each resource gets a unique and resolvable address in the form of a URL. Thus, the first step a ROA design is to **identify the resources** an EPCIS should be composed of and to make them **addressable**. Looking at the EPCIS standard, we can extract a dozen resources. We focus here on the four main types:

1. Locations (called "Business locations" in the EPCIS standard): those are locations where events can occur, e.g.,:"C Floor, Building B72".
2. Readers (called "ReadPoints" in the standard): which are RFID readers registered in the EPCIS. Just as Business Locations, readers are usually represented as URIs: e.g., `urn:br:maxhavelaar:natal:shipyear:incoming` but can also be represented using free-form strings, e.g.,: "Reader Store Checkout"
3. Events: which are observations of RFID tags, at a Business Location by a specific reader at a particular time.
4. EPCs: which are Electronic Product Codes identifying products (e.g., `urn:epc:id:sgtin:618018.820712.2001`), types of products (e.g., `urn:epc:id:sgtin:618018.820712.*`) or companies (e.g., `urn:epc:id:sgtin:618018.*`).

We first define a hierarchical organization of resources based on the following URI template:

```
location/businessLocation/reader/readPoint/time/
eventTime/event
```

More concretely, this means that the users begin by accessing the Location resources. Accessing the URL `http://.../location/` with the `GET` method retrieves a list of all Locations currently registered in the EPCIS. From there, clients can navigate to a particular Location where they will find a list of all Readers at this

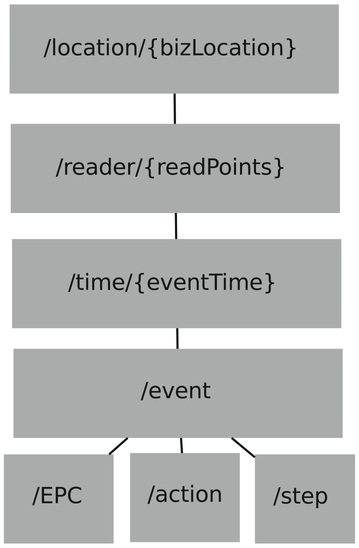


Fig. 16.2 Hierarchical representation of the browsable RESTful EPCIS resources

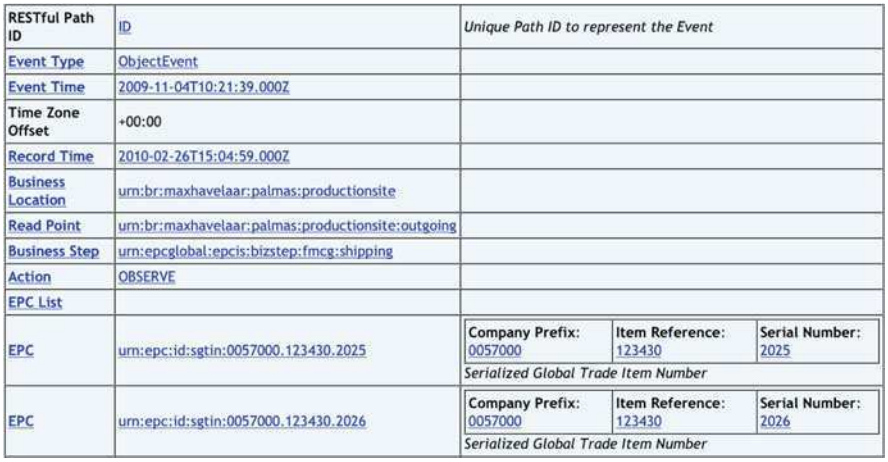
place. From the Readers clients get access to Time resources which root is listing all the Times at which Events occurred. By selecting a Time, the client finally accesses a list of Events.

Each event contains information such as its type, event time, Business Location, EPCs, etc. If a client is only interested about one specific field of an Event, he can get this information by adding the desired information name as sub-path of the Event URI. For example, `EVENT_URI/epcs` lists only all the EPCs that were part of that Event. The resulting tree structure is shown in Fig. 16.2, and a sample Event in Fig. 16.3.

Furthermore, in a ROA all resources should be discoverable by browsing to facilitate the integration with the Web. Just as you can browse for Web pages, we should be able to find RFID tagged objects and their traces by browsing. Each representation of resources should contain links to relevant resources such as parents, descendants or simply related resources. This property of ROAs is known as "connectedness".

To ensure the connectedness of the RESTful EPCIS, each resource in the tree links to the resources below or to related resources. The links allow users to browse completely through the RESTful EPCIS where links act as the motor. Every available action is deduced by the set of links included. This way, people can directly explore the EPCIS from any Web browser, simply by clicking on hyperlinks and without requiring any prior knowledge of the EPCIS standard.

To ensure that the browsable EPCIS interface did not become too complicated, we limited the number of available resources and parameters. For more complex queries we provide a second, hierarchical, interface for which we map the EPCIS



RESTful Path ID	ID	*Unique Path ID to represent the Event*
Event Type	ObjectEvent	
Event Time	2009-11-04T10:21:39.000Z	
Time Zone Offset	+00:00	
Record Time	2010-02-26T15:04:59.000Z	
Business Location	urn:br:maxhavelaar:palmas:productionsite	
Read Point	urn:br:maxhavelaar:palmas:productionsite:outgoing	
Business Step	urn:epcglobal:epcis:bizstep:fmcg:shipping	
Action	OBSERVE	
EPC List		
EPC	urn:epc:id:sgtin:0057000.123430.2025	Company Prefix: 0057000 / Item Reference: 123430 / Serial Number: 2025 / *Serialized Global Trade Item Number*
EPC	urn:epc:id:sgtin:0057000.123430.2026	Company Prefix: 0057000 / Item Reference: 123430 / Serial Number: 2026 / *Serialized Global Trade Item Number*

Fig. 16.3 HTML representation of an EPC event as rendered by a Web browser, every entry is also a link to the sub-resources

WS-* query interface to uniquely identifiable URIs. Each query parameter can be encoded and combined as a URI query parameter according to the following template

```
/eventquery/result?param1=value1&...&paramN=valueN
```

Query parameters restrict the deduced result set of matching RFID events. The RESTful EPCIS supports the building of such URIs with the help of an HTML form. If for example a product manager from Max Havelaar is interested in the events that were produced in Palmas, the following URL lists all events that occurred at this business location:

```
http://.../eventquery/result?location=urn:br:
maxhavelaar:palmas:productionsite
```

To further limit possibly very long search results, the query URI can be more specific. The manager might be interested only about what happened on that production site on the 4th of November 2009, which corresponds to the following URL:

```
http:/../eventquery/result?location=urn:br:
maxhavelaar:palmas:productionsite&time=2009-11-04T00:
00:00.000Z,2009-11-04T 23:59:59.000Z
```

The HTML representation of this resource is illustrated in Fig. 16.3.

To keep the full connectedness of the RESTful EPCIS, both the browsable and the query interface are interlinked. For example, the EPC `urn:epc:id:sgtin:0057000.123430.2025` included in the event of Fig. 16.3, is also a link to the query which asks the EPCIS for all events that contain this EPC.

We leverage the addressability property to allow a greater interaction with EPCIS data on the Web. As an example, since queries are now encapsulated in URLs, we can simply bookmark them, exchange them in emails and consume them from

JavaScript applications. Furthermore, by implementing the connectedness property we enable users to discover the EPCIS content in a simple, yet powerful manner.

Uniform Interface and Self-Describing Messages

Finally, in a ROA, the resources and their services should be accessible using a standard interface defining the mechanisms of interaction. The Web implementation of REST uses HTTP for this purpose.

Multiple Representation Formats A resource is representation agnostic and hence should offer several representations (e.g., XML, HTML). HTTP provides a way for clients to retrieve the most adapted one. The RESTful EPCIS supports multiple output formats to represent a resource. Each resource first offers an HTML representation as shown in Fig. 16.3 which is used by default for Web browser clients.

In addition to the HTML representation, each resource also has an XML and a JSON (JavaScript Object Notation) representation, which all contain the same information. The XML representation complies with the EPCIS standard and is intended to be used mainly for business integration. The JSON representation can be directly translated to JavaScript objects and is thus intended for mashups, mobile applications or embedded computers.

The choice of the representation to use in the response can be requested by clients using the HTTP "content negotiation" mechanism.[3] Since content negotiation is built into the uniform interface, clients and servers have standardized ways to exchange information about available resource representations, and the negotiation allows clients and servers to choose the representation that fits best a given scenario.

A typical content negotiation procedure looks s follows. The client begins with a `GET` request on `http://.../location`. It also sets the `Accept` header of the HTTP request to a weighted list of media types it can understand, for example to: `application/json`, `application/xml;q=0.5`. The RESTful EPCIS then tries to serve the best possible format it knows about and describes it in the `Content-Type` of the HTTP response. In this case, it will serve the results in the JSON format as the client prefers it over XML ($q = 0.5$).

Error Codes The EPCIS standard defines a number of exceptions that can occur while interacting with an EPCIS. HTTP offers a standard and universal way of communicating errors to clients by means of "status codes". Thus, to enable clients, especially machines to make use of the exceptions defined by the EPCIS specification, the RESTful EPCIS maps the exceptions to HTTP status codes. An exhaustive list of error codes and their meanings for Resource Oriented Architectures can be found in Richardson and Ruby (2007).

[3]http://www.w3.org/Protocols/rfc2616/rfc2616-sec12.html.

Syndication with Atom

In many cases, it would be useful to group tagged objects into collections according to certain properties or scenarios (example collections would be “all the milk bottles shipped today to rhode island” or “potatoes shipped to client no 3”), and be able to monitor the state of collection through a syndication mechanism. The Atom Syndication Format is an XML language specifying the syntax of Web feeds. With Atom, the Web has a standardized and RESTful model for interacting with collections, and the *Atom Publishing Protocol (AtomPub)* extends Atom’s read-only interactions with methods for write access to collections. Because Atom is RESTful, interactions with Atom feeds can be based on simple `GET` operations which can then be cached.

Case Study: Web-Enabling the Subscriptions

Standard EPCISs also offers an interface to subscribe to RFID events. Through a WS-* operation, clients can send a query along with an endpoint (i.e., a URL) and subscribe for updates. Every time the result of the query changes, an XML packet containing the new results is sent to the endpoint. While this mechanism is practical, it requires for clients to run a server with a tailored Web applications that listens to the endpoint and thus cannot be used by all users or cannot be directly integrated to a Web browser.

This makes the subscription interface an ideal candidate to apply the idea of Web feeds with Atom. Thus, in the RESTful EPCIS, we propose an alternative Atom module for producing the results of query subscriptions as shown on the leftmost side of Fig. 16.4. This way, end-users can formulate queries by browsing the RESTful EPCIS and get updates in the Atom format which most browsers can understand and directly subscribe to.

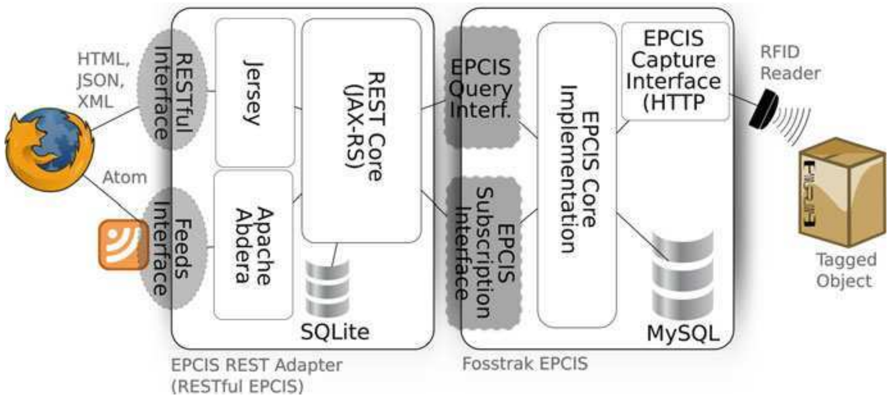


Fig. 16.4 Architecture of the RESTful EPCIS based on the Jersey RESTful framework and deployed on top of the Fosstrak EPCIS

As an example a product manager could create a feed in order to be automatically notified in his browser or any feed reader whenever one of his products is ready to be shipped from the warehouse. More concretely, this can be done by sending the following HTTP `PUT` request:

```
http://.../eventquery/subscription?reader=urn:ch:migros:
stgallen:warehouse:expedition&epc=urn:epc:id:sgtin:
0057000.123430.*
```

Or, for a human client, clicking on the "subscribe" link present at the top of each HTML representation of query results. As a result, the RESTful EPCIS will create an Atom feed corresponding to this query and add an entry (using AtomPub) to the feed every time an event for the product category `123430` is generated by reader `urn:ch:migros:stgallen:warehouse:expedition`.

The product manager can then use the URI of the feed in order to send it to his customers, allowing them to follow the goods progress as well. A simple but very useful interaction which would require a dedicated client to be developed and installed by each customer in the case of the WS-* based EPCIS.

Implementing RESTful Information Systems

After the design of RESTful Services, comes their implementation. The recent regain of interest for RESTful services has led to a number of frameworks helping developers in this step. In this section we will look at some of these frameworks, focusing on their features and benefits when applying the constraints of RESTful architectures. However, let us begin by looking at integration patterns at a higher level: given an existing information system, what integration options do we have?

From WS- to REST: Integration Patterns*

When creating an information system from scratch, the constraints for RESTful architectures are of great help in defining the data model. There are also no major conflicts between the REST paradigm and the Object Oriented paradigm. Indeed, Object Oriented programming defines an internal, application centric, contract. REST, on the other hand, defines a contract with the world outside the application (this is why developers often speak about RESTful APIs) towards a distributed and remote usage of its functionality. Thus, both can cohabit nicely to create a distributed Web application, as long as they are designed together. However, adding a RESTful architecture to an existing WS-* centric information system can be challenging as both paradigms share the same basic goal: creating remotely re-usable services.

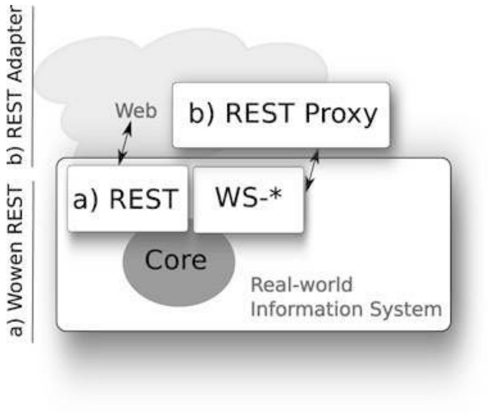


Fig. 16.5 Integration patterns for adding a RESTful interface to a WS-* system

Woven REST

As shown on Fig. 16.5, there are basically two ways of achieving an integration; First (a) on Fig. 16.5), the RESTful architecture can be directly woven into the existing WS-* system. This may seem like a trivial solution at first; however, the implementation of this solution is not entirely straightforward. While sharing a common goal, WS-* and REST are rooted on very different paradigms. Thus, weaving clean REST architecture into the core of the WS-* system almost always requires an alternate data model. Using two different data models for the same services ends up in rather complicated architectures.

REST Adapter

An alternative integration pattern is to design an external REST Adapter making use of the WS-* interface, as shown in (b) of Fig. 16.5, REST Adapter. In this model, the REST Adapter acts as a proxy, translating RESTful requests into WS-* requests. This allows for a cleaner, REST centric architecture and preserves the legacy WS-* system entirely intact. On the downside it hinders the performances of the RESTful API but, as we will show in the case study, this can be minimized to a level acceptable for most applications.

Case-study: RESTful EPCIS as a Module

For the RESTful EPCIS, we created an independent REST Adapter, as it delivers a clear advantage in this case: it allows the RESTful EPCIS to work on top of any standard EPCIS implementation.

The resulting architecture is shown in Fig. 16.4. The RESTful EPCIS is a module which core is using the EPCIS WS-* standard interface. Just as a proxy, it translates the incoming RESTful request into WS-* requests and returns results complying

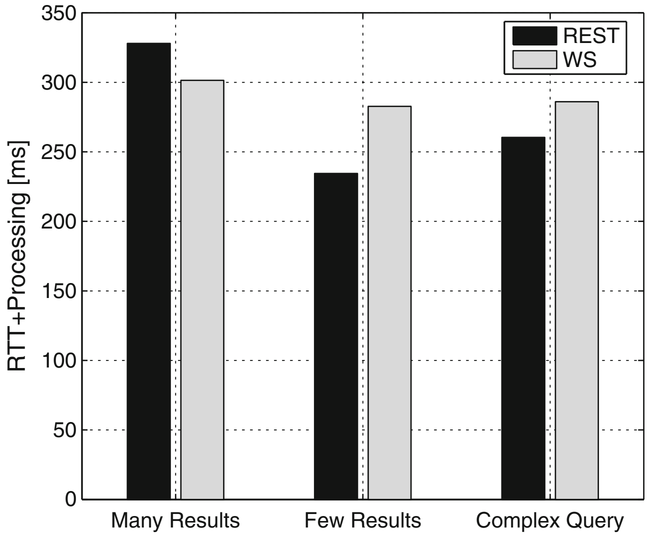


Fig. 16.6 Average RTT and processing time when using the WS-* interface and the REST interface for three types of requests each run 100 times

with the constraints of RESTful architectures. As shown on the left of the picture, the typical clients of the RESTful EPCIS are different from the business applications traditionally connected to the EPCIS. The browser is the most prevalent of these clients. It can either directly access the data by means of URL calls or indirectly using scripted Web pages.

Performance Evaluation

As mentioned before, the translation between REST and WS-* (and vice-versa) results in an overhead that we briefly evaluate here.

The experimental setup is composed of a Linux Ubuntu Intel dual-core PC 2.4 GHz with 2 GB of ram. We deploy Fosstrak and the RESTful EPCIS on the same instance of Apache Tomcat with a heap size of 512 MB. We evaluate three types of queries all returning the standard EPCIS XML representation.

The first query (Q1, "Many Results" in Fig. 16.6) requests all events recorded by the EPC, i.e., a small request returning a document of 30 KB with 22 events each composed of about 10 EPCs. In the second test (Q2, "Few Results"), is a query returning a document of 2.2 KB with only two results. The last test (Q3, "Complex Query") is a query containing a lot of parameters and returning ten events. We test each of these queries asking for the standard XML representation. All queries are repeated 100 times from a client located on a machine one hop away from the server with a Gigabit ethernet connectivity. The client application is programmed in Java and uses a standard JAX-WS client for the WS-* calls and the standard Apache HTTP Client and DOM (Document Object Model) library for the REST calls.

As shown on Fig. 16.6, for Q1 the RESTful EPCIS has an average overhead of 30 ms due to the computational power required to translate the requests from REST to WS-* and vice-versa. For Q2 and Q3 the REST requests are executed slightly faster (about 20 ms) than the WS-*. This is explained by three factors. First, since there are fewer results, the local WS-* request from the RESTful EPCIS is executed faster. Then, REST packets are slightly smaller as there is no SOAP envelope (Yazar and Dunkels 2009). Finally, unmarshalling WS-* packets (using JAXB) on the client-side takes significantly longer than for REST packets with DOM. For Q3, similar results are observed. Overall, we can observe that the RESTful EPCIS creates a limited overhead of about 10% which is compensated in most cases by the relatively longer processing times of WS-* replies. This becomes a particularly important point when considering devices with limited capabilities such as mobile phones or sensor nodes as well as for client-side (e.g., JavaScript) web applications.

It is worth mentioning that the WS-* protocol can be optimized in several ways to better perform, for example by compressing the SOAP packets and optimizing JAXB. However as the content of HTTP packets can also be compressed this is unlikely to drastically change the results. Furthermore, because they encapsulate requests in HTTP `POST`, WS-* services cannot be cached on the Web using standard mechanisms. For the RESTful EPCIS however, all the queries are formulated as HTTP `GET` requests and fully contained in the request URL. This allows to directly leverage from standard Web caching mechanisms (Fielding and Taylor 2002) which would importantly reduce the response times (Yazar and Dunkels 2009).

Understanding the Tools Galaxy in Java

Creating clients for RESTful Web Services is a rather straightforward task as it only requires for the used language to support HTTP, which most modern programming and scripting languages do. The implementation of a RESTful Web Services, on the other hand, is a task that should not be underestimated. Indeed, even if the set of REST constraints is seemingly small their implementation requires a careful software design.

Most modern Web languages such as Ruby (especially in its Ruby on Rails form) or Python offer out-of-the-box support for RESTful Web Services. Similarly, the recent growing interest for lightweight service architectures based on REST has given birth to a number of frameworks that simplify the development of RESTful applications for enterprise-scale languages such as C# or Java.

JAX-RS: A Standard Java API for RESTful Web Services

The Java community is a particularly interesting one since it is known as one of the community with most WS-* tools and frameworks but also as one of the most eager to develop tools around REST (perhaps due to some frustrations with the WS-* type of services...).

In particular, the Java galaxy has its own higher-level industrial standard for building RESTful Web Services: the JAX-RS API[4] (also known as JSR 311). JAX-RS is especially interesting since it was developed by a consortium of people who are both Web-specialists and service developers. The result is a very lean API [well described in Burke (2009)] that requires a good understanding of REST but offers straightforward solutions to implement in an elegant and efficient way most of the REST constraints.

In short, JAX-RS is based on three main pillars. It first uses annotations of Java classes to turn them into resources (e.g., `@Path(''/location'')`), ensuring the *Resource Identification* constraint. Annotations further help to define the resources' *Uniform Interface* as it lets the developer specify allowed verbs (`@GET`, `@POST`) and served representations (e.g., `@Produces(MediaType.APPLICATION_JSON)`). Beyond annotations, several framework classes make the developer life easier. *Connectedness* is boosted by providing contextual `URI Builders`, letting the developer easily link resources together across representation. Finally, the use of the JAXB framework allows for Java Objects to be automatically serialized to an (extensible) number of representations such as XML, HTML, JSON and Atom thus making it easier to fulfill the constraint for *Self-Describing Messages*.

Besides Jersey,[5] the reference implementation of JAX-RS, several frameworks such as RESTeasy, Apache Wink, Apache CFX and RESTlet are JAX-RS compliant which makes it rather easy to move code from one framework to the other.

Case-study: Using JAX-RS, Jersey and Abdera

As shown in Fig. 16.4, the core of the RESTful EPCIS is based on the JAX-RS compliant, Jersey[6] framework. Thus, it uses JAX-RS annotations and framework classes. The example below serves the representation of a `location` resource.

```
@Path(\location\{businessLocationID})
@GET
@Produces({MediaType.APPLICATION_XML, MediaType.
    APPLICATION_JSON, MediaType.APPLICATION_ATOM_XML,
    MediaType.TEXT_HTML})
public Resource getSelectedBusinessLocation(@Context
    UriInfo context, @PathParam("businessLocationID")
    String businessLocation) {
    QueryBusinessLogic logic = new QueryBusinessLogic();
    return logic.getSelectedBusinessLocation(context,
        businessLocation);
}
```

[4] http://jcp.org/en/jsr/detail?id=311.

[5] http://https://jersey.dev.java.net.

[6] https://jersey.dev.java.net.

Line 1 of this listing sets the URI of the resource, where `businessLocationID` is the `location` identifier which will be dynamically passed to the method `getSelectedBusinessLocation` at runtime. `@GET` specifies the method allowed on this resource, `@Produces` contains the representations that clients will be able to obtain through content negotiation. Note that these contents will be automatically generated at runtime from the `Resource` Java Object by the JAXB framework.

As we can see, the RESTful EPCIS uses Jersey for managing the resources' representations and dispatching HTTP requests to the right resource depending on the request URL. When correctly dispatched to the RESTful EPCIS Core, every request on the querying or browsing interface is then translated to a WS-* request on the EPCIS. This makes the RESTful EPCIS entirely decoupled from any particular implementation of an EPCIS.

While JAX-RS offers serving Atom representation of resources on-the-fly, implementations of JAX-RS do not have to offer a fully-featured Atom-Pub server with persistence. Thus, for the subscription interface we used Apache Abdera, which is an open-source implementation of an Atom-Pub server integrating well with most JAX-RS frameworks. Every time a client subscribes to a query, the RESTful EPCIS checks whether this feed already exists by checking the query parameters, in any order. If it is not the case it creates a query on the WS-* EPCIS and specifies the address of the newly created feed. As a consequence every update of the query is directly `POST`ed to the feed resource which creates a new entry using Abdera and stores it in an embedded SQLite[7] database.

Jersey, Abdera and SQLite are packaged with the RESTful EPCIS core in a Web Application Archive (WAR) that can be deployed in any Java compliant Web or Application Server. We tested it successfully on Glassfish[8] and Apache Tomcat[9] and on the Grizzly embedded Web Server.[10]

REST and the Mashups

As RFID objects become part of the Web, applications using them can be developed using popular Web languages (e.g. HTML, JavaScript, PHP, Python) and toolkits, (e.g., DOJO, jQuery, Closure). This can significantly ease the developments on the RFID middleware vendor's side, since applications can be built on languages for which a plethora of libraries and toolkits are available. Furthermore, the use of popular languages makes it easier to find adequate developers. Likewise, this also unveils the possibility for external developers to create innovative Web applications

[7]http://www.sqlite.org.

[8]http://glassfish.org.

[9]http://tomcat.apache.org.

[10]http://grizzly.dev.java.net.

making use of RFID data. Open APIs and communities of developers have long become vital for service companies on the Web such as Facebook, Twitter, or Google. This direction is also being taken upon by many electronic devices (sensor nodes, appliances, etc.). New hardware on the market such as the Chumby alarm clock[11] or the Squeezbox HiFi system[12] already have significant communities of voluntary Web developers creating dozens of small applications for each platform.

Adding a RESTful module to the EPCIS brings it one step closer to these promising opportunities, where the consumers become active actors, not just passive consumers. Just as users create Web 2.0 mashups (Yu et al. 2008) by integrating several Web sites to create new applications, companies buying RFID systems can re-use RFID events to create ad-hoc, innovative applications in an easier manner. The EPCIS RESTful API allows a wider range of developers, tech-savvy users (technologically skilled people) or researchers to develop on top of the EPCIS and contributes to helping the EPC Network developer community grow.

Case Study: The EPC Dashboard Mashup

To better illustrate the new type of applications the RESTful EPCIS unveils, we created the EPC Dashboard Mashup, a Web mashup, that helps product, supply chain and store managers to have a live overview of their business at a glance. It can further help consumers to better understand where the goods are coming from and what other people think about them. The EPC Dashboard is based on the concept of widgets in which the events data are visualized in a relational, spacial or temporal manner.

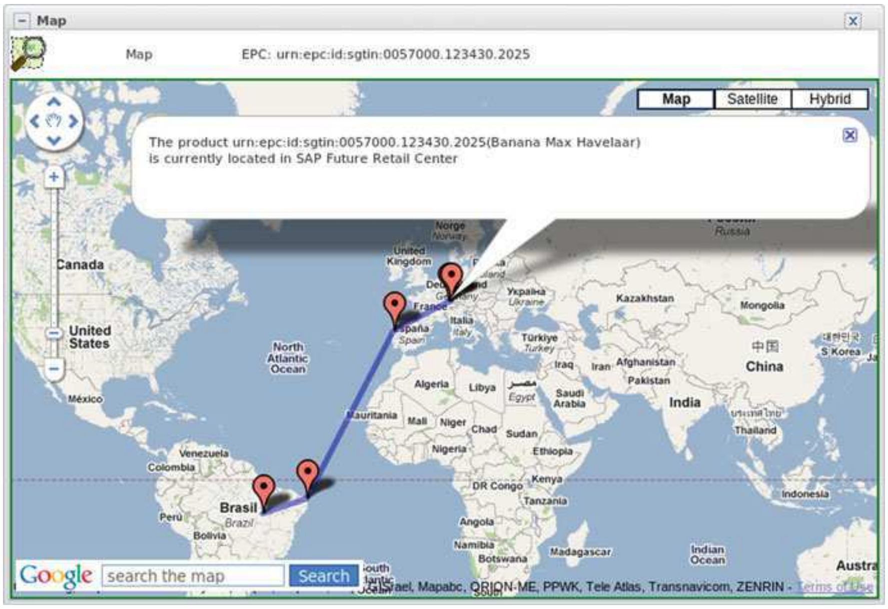
The EPC Dashboard consumes data from the RESTful EPCIS. Usually these data are hard to interpret and integrate. The dashboard makes it simple to browse and visualize the EPC data. Furthermore, it integrates the data with multiple sources on the Web such as Google Maps, Wikipedia, Twitter, etc.

Mashup Architecture

The EPC Dashboard integrates several information sources. This information is encapsulated in small windows called widgets. The widgets combine services on the Web with traces coming from the RESTful EPCIS. The EPC Dashboard Mashup currently offers 12 widgets using different APIs and services. As an example, the Map Widget is built using the Google Maps Web API (see Fig. 16.7), the Product Buzz Widget uses the Twitter RESTful API (Fig. 16.8) and the Stock History Widget uses the Google Visualization API.

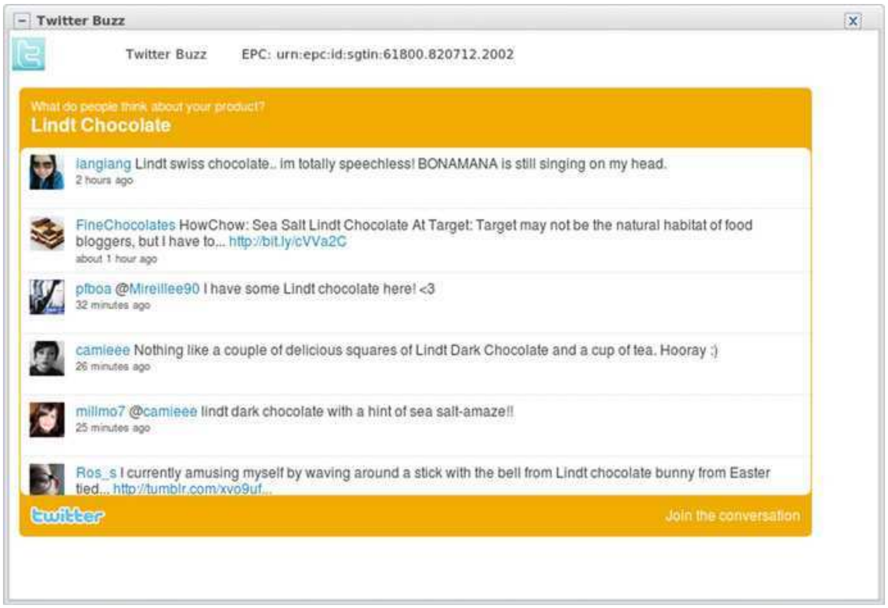
[11]http://www.chumby.com.

[12]http://www.logitechsqueezebox.com.



The product urn:epc:id:sgtin:0057000.123430.2025(Banana Max Havelaar) is currently located in SAP Future Retail Center

Fig. 16.7 The Maps widget is following the route of the banana tagged with the EPC urn:epc:id:sgtin:0057000.123430.2025



What do people think about your product?
Lindt Chocolate

iangiang Lindt swiss chocolate.. im totally speechless! BONAMANA is still singing on my head.
2 hours ago

FineChocolates HowChow: Sea Salt Lindt Chocolate At Target: Target may not be the natural habitat of food bloggers, but I have to... http://bit.ly/cVVa2C
about 1 hour ago

pfboa @Mireillee90 I have some Lindt chocolate here! <3
32 minutes ago

camieee Nothing like a couple of delicious squares of Lindt Dark Chocolate and a cup of tea. Hooray :)
26 minutes ago

millmo7 @camieee lindt dark chocolate with a hint of sea salt-amaze!!
25 minutes ago

Ros_s I currently amusing myself by waving around a stick with the bell from Lindt chocolate bunny from Easter tied... http://tumblr.com/xvo9uf...

Fig. 16.8 The Product Buzz Widget extracts live opinions and information about particular products (here Lindt Chocolate) from Twitter

All widgets are connected to each other which means that actions on a given one can propagate the selection to the other widgets and changes their view accordingly. As such, widgets listen to selections and can make selections. This interaction is implemented using the observer pattern (Gamma et al. 1994) where consumers (i.e., the widgets) register to asynchronous updates of the currently selected Locations, Readers, Time or EPCs. This architecture allows the creation and integration of other Web widgets with very little effort. The EPC Dashboard itself is a JavaScript application built using the Google Web Toolkit,[13] a framework to develop rich Web clients. This has been possible because having a RESTful Interface upon the EPCIS eases the development of mashups.

Summary

In this chapter we argue that RESTful architecture can greatly contribute to the success and public innovation around an Information System. We further argue for thinking of these systems as Web APIs rather than as applications. As an illustration we describe how we applied the principles and constraints of RESTful architectures to the world of RFID for creating the RESTful EPCIS open-source project which is released as an open-source module of the Fosstrak project, under the name of epcis-restadapter.[14]

RESTifying the EPCIS literally brings RFID traces to the Web, as every tagged product, reader, location, etc. become fully addressable resources. Using the HTTP protocol tagged objects can be directly searched for, indexed, bookmarked, exchanged and feeds can be created by end-users. Furthermore, this enables exploring the EPCIS data simply by browsing them, which helps making sense of the data. We argue that this adds more flexibility to the types of applications that can be built on top of an EPCIS and opens the EPCIS API for fast-prototyping to the very large and active community of Web and mobile developers. We further show that this added flexibility does not necessarily have to hinder the overall performances, deploying the RESTful EPCIS on the same machine as the WS-* EPCIS leads to satisfactory results while preserving the EPCIS-vendor independence.

We finally illustrate the new application space the RESTful EPCIS unveils by means of a JavaScript Mashup: the EPC Dashboard which is an easily extensible business intelligence interface for product managers that re-uses a number of Web APIs.

[13] http://code.google.com/intl/en/webtoolkit.

[14] http://www.webofthings.com/rfid.

References

Bill Burke. *RESTful Java with Jax-RS*. O'Reilly Media, 1st edition, November 2009.

Roy T. Fielding and Richard N. Taylor. Principled design of the modern Web architecture. *ACM Trans. Internet Technol.*, 2(2): 115–150, 2002.

Christian Floerkemeier, Matthias Lampe, and Christof Roduner. Facilitating RFID Development with the Accada Prototyping Platform. In *Proceedings of the Fifth IEEE International Conference on Pervasive Computing and Communications Workshops*, pages 495–500. IEEE Computer Society, Silver Spring, MD, 2007.

Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, Reading, MA (Wokingham, UK), November 1994.

Dominique Guinard, Vlad Trifa, and Erik Wilde. A Resource Oriented Architecture for the Web of Things. In *Proceedings of IoT 2010 (IEEE International Conference on the Internet of Things)*, Tokyo, Japan, November 2010.

Dominique Guinard, Felix von Reischach, and Florian Michahelles. MobileIoT Toolkit: Connecting the EPC Network to MobilePhones. In *Proceedings of Mobile Interaction with the Real World at Mobile HCI (MIRW)*, The University of Oldenburg, Amsterdam, Netherlands, September 2008.

Tim Kindberg, John Barton, Jeff Morgan, Gene Becker, Debbie Caswell, Philippe Debaty, Gita Gopal, Marcos Frid, Venky Krishnan, Howard Morris, John Schettino, Bill Serra, and Mirjana Spasojevic. People, places, things: web presence for the real world. *Mob. Netw. Appl.*, 7(5): 365–376, 2002.

T. Luckenbach, P. Gober, S. Arbanowski, A. Kotsopoulos, and K. Kim. TinyREST – A protocol for integrating sensor networks into the internet. In *Proceedings of the Workshop on Real-World Wireless Sensor Network (SICS)*, Stockholm, Sweden, 2005.

Cesare Pautasso and Erik Wilde. Why is the Web Loosely Coupled? A Multi-faceted Metric for Service Design. In *Proceedings of the 18th International World Wide Web Conference (WWW'09)*, Madrid, Spain, April 2009.

Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful Web Services vs. Big Web Services: Making the Right Architectural Decision. In *Proceedings of the 17th International Conference on World Wide Web (WWW)*, pages 805–814, ACM, New York, NY, USA, 2008.

Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly Media, Inc., May 2007.

Dogan Yazar and Adam Dunkels. Efficient Application Integration in IP-based Sensor Networks. In *Proceedings ACM of the First ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings (BuildSys)*, Berkeley, CA, USA, November 2009.

Jin Yu, Boualem Benatallah, Fabio Casati, and Florian Daniel. Understanding Mashup Development. *IEEE Internet Comput.*, 12(5): 44–52, 2008.

Chapter 17
Leveraging the Web for a Distributed Location-aware Infrastructure for the Real World

Vlad Trifa, Dominique Guinard, and Simon Mayer

Abstract Since GPS receivers have become a commodity anyone could access and use location information simply and freely. Such an easy access to one's location is instrumental to the development of location-aware applications. However, existing applications are static in that they do not model relations between places and mobile things. Moreover, these applications do not allow to easily map the physical location of mobile devices to virtual resources on the Internet. We attempt to bridge this gap by extending the base concepts that make up the Internet with the physical location of devices, in order to facilitate the development of Web-based location-aware applications for embedded mobile devices. In this chapter, we propose a simple infrastructure for the "Web of Things" that extends the existing Web to enable location-aware applications. The proposed solution enables a naturally hierarchic way to search for location-aware devices and the services they provide.

Introduction

In the last decade, tiny computers with various onboard sensors have been increasingly installed in our buildings and cities. Connecting all these sensors to a unique infrastructure has the potential to significantly affect our daily lives by facilitating access to massive amounts of real-time data and reacting rapidly to various conditions. For example, a manufacturing company could monitor and detect events or anomalies in the production line rapidly, thus could react and prevent stops of production or even accidents by having an instant view of what is happening across the various locations of the company at any given time.

V. Trifa (✉)
Institute for Pervasive Computing – ETH Zurich, Universitätstrasse 6, 8092 Zurich, Switzerland
e-mail: trifa@acm.org

As the usage of such networked sensing devices will spread, efficient – yet simple – mechanisms and tools for automated data acquisition and manual interaction or control will be increasingly required. As more and more devices will need to interact and work with each other in an ad hoc manner, an interoperable and open infrastructure for seamless integration and usage of devices will become a necessity. Recent efforts in the Web of Things (WoT) (Guinard et al. 2010) domain have shown that REST (Fielding and Taylor 2002) is an appropriate architectural style for building pervasive computing applications. Various prototypes have illustrated the advantages associated with the use of Web technologies for ad hoc interaction with devices. However, the lack of a scalable and flexible infrastructure to support and automate the search and discovery of devices based on their characteristics represents a major obstacle when building large-scale applications on top of thousands of heterogeneous and mobile sensing devices.

At any given time, any person or object has a unique location in the physical world (home, office, car, etc.). In contrast, the physical location of data is irrelevant on the Web, since an efficient mechanism (universal resource identifiers, URI) is in place to access data regardless of where it is actually stored. For the Web to truly embrace the physical world, one needs to extend the classic Web model to make it easy to bind real-time contextual information to things and use this information to search things. The centralized index approach commonly used by search engines seems appropriate for storing massive amounts of historical data. However, when it comes to monitoring millions (or billions) of resources that will form the Web of Things, a radically different approach is required. As more and more things will have to be monitored in real-time, a centralized repository to store and query their status would hardly scale. Present-day location-aware services such as Gowalla[1] or Foursquare[2] are nothing more than classical Web applications, therefore direct, ad hoc interaction with physical places and their services is impossible without being mediated through the remote server.

Although the cheap GPS receivers embedded in mobile phones have played a central role in the popularization of such location-aware applications, they are not useable when it comes to indoor localization. Because it does not rely on an expensive or dedicated infrastructure, Wi-Fi fingerprinting is becoming a particularly viable alternative that works at a city-scale and even indoors. With an accuracy of a few meters, room-level localization is reasonably feasible which is sufficient for most ubiquitous computing applications (Abowd et al. 2000). Even though spatial localization techniques are constantly improving open and physically distributed location-aware applications are still prevented by the lack of robust and open standards for modeling and representing locations on the Web in a more flexible format than geographical coordinates (Wilde and Kofahl 2008). Due to the lack of tools and techniques to support natively the physical location of things on the Web, discovering devices present in a certain place and interacting with

[1]http://gowalla.com/.

[2]http://foursquare.com/.

them directly in an ad hoc manner (i.e., without mediation through a centralized repository) is a complex problem which still requires custom applications and protocols. This is further aggravated by the multitude of incompatible protocols for low-power devices that coexist today. While solutions such as Bluetooth, Apple's Bonjour or Universal Plug and Play do offer powerful mechanisms for locating devices on a network, they remain overly complex, are incompatible with Web technologies, and do not support the physical location of devices.

In this chapter, we describe InfraWoT, a possible solution for these problems that builds on top of state-of-the-art research in the Web of Things. We show why a RESTful architecture is an ideal solution for leveraging an existing Wi-Fi infrastructure to build a loosely-coupled infrastructure for searching and interacting with networked devices depending on their physical location. Even though the Web was designed as a hyper-linked system for multimedia documents, this chapter shows that a distributed location-aware infrastructure for embedded devices can be built solely using Web standards. In particular, we discuss how REST can be leveraged to simplify ad hoc interactions with devices by considering the spatial relations between places, devices, and people.

We extend the concept of *gateways* proposed in earlier work to connect the Web with the real world by enabling RESTful interactions with embedded devices and sensors (Trifa et al. 2009). By linking physically distributed gateways on the Web, we can form self-stabilizing hierarchical structures (trees) that can be mapped to physical locations and symbolic place concepts such as buildings, floors, rooms, etc. When new devices connect to this network through a gateway, they inherit automatically the location of the gateway they connect to, which enables physical objects to be searched, accessed, browsed, and linked together just like any other Web resource. On top of this hierarchical place model, we illustrate how Web clients can use the HTTP/URI mechanism as a lightweight and simple, yet powerful, flexible, and expressive combo to perform context-aware searches to find and use relevant objects at specific locations in real-time.

A Web-oriented Infrastructure for Physical Things

The success of the World Wide Web stems from its particular software architecture called Representational State Transfer (REST), which emphasizes scalability, generic interfaces, and a loose coupling between components (Fielding and Taylor 2002). On the Web, the primary abstraction of information and functionality are *resources* that are identified by Uniform Resource Identifiers (URIs) and can be interacted with using the HTTP protocol. Although HTTP was designed as an application protocol with particular strengths (and weaknesses), many Web applications today reduce its role to a mere transport protocol by using only a fraction of its features. For example, Web applications that rely upon Web services based on SOAP and WSDL use only a single operation of HTTP (POST) to call API methods offered by a few URI-identified endpoints, thus hiding the actual resources being manipulated. This prevents to take full advantage of the Web architecture's features and tools (e.g., caching, load balancing, etc.), as it requires to define specific application layers for each application.

Web-enabling Things

The term *Internet of Things (IoT)* refers to networked devices with an emphasis on interoperability at the data transport layer to maximize raw performance through customized, tightly-coupled applications. More recently, the *Web of Things (WoT)* (Guinard et al. 2010) vision proposed a shift towards simplified integration and programming by reusing well-known Web standards to interact with embedded devices. This way, common Web tools (e.g., browsers), interaction techniques, and languages can be directly used to program the physical world. Following the success of Web 2.0 mashups, we suggest that a similar lightweight approach for interacting with embedded devices using HTTP to manipulate URI-identified resources, significantly reduces the time required to develop applications for devices and enables the creation of *physical mashups* (Guinard and Trifa 2009).

Another advantage of Web protocols over lower-level Internet protocols when connecting smart real-world devices to the cyberspace is that one inherits many of the mechanisms that made the Web scalable and successful for example caching, load balancing, indexing, and searching as well as the stateless nature of the HTTP protocol. One can also leverage search engines to register and index a physical service (e.g., monitoring environmental sensors), by using semantic annotations to describe the functionality and interaction possibilities of each device.

Embedded devices usually have limited resources and therefore require optimized protocols to exchange data. Additionally, as HTTP or IP might not be available or appropriate for such devices, we propose to use gateways to enable Web-based interactions with low-power devices. Such a *gateway* (cf. Fig. 17.1) is nothing more than a Web application that enables access to heterogenous devices through a simple and uniform RESTful API, thus hiding the complexity of the various protocols used by the devices (such as Bluetooth or Zigbee). The gateway application is lightweight enough to run on any programmable computer with a TCP/IP connection that can run Java, as for example programmable wireless routers, network-attached storage (NAS) devices, or networked media players.

Search engines have allowed to index and search the whole Web, we believe that the next evolution will be the search for real-time data in the physical world. As demonstrated by the success of the Web, a loosely-coupled physically distributed application can scale massively. To replicate this characteristic, in this chapter we explore how one can bind gateways and their associated devices together to form a large infrastructure that integrates all kinds of devices over the Web. Such an infrastructure could enable to search and use devices according to their current state, location or overall context on a global scale and in real time.

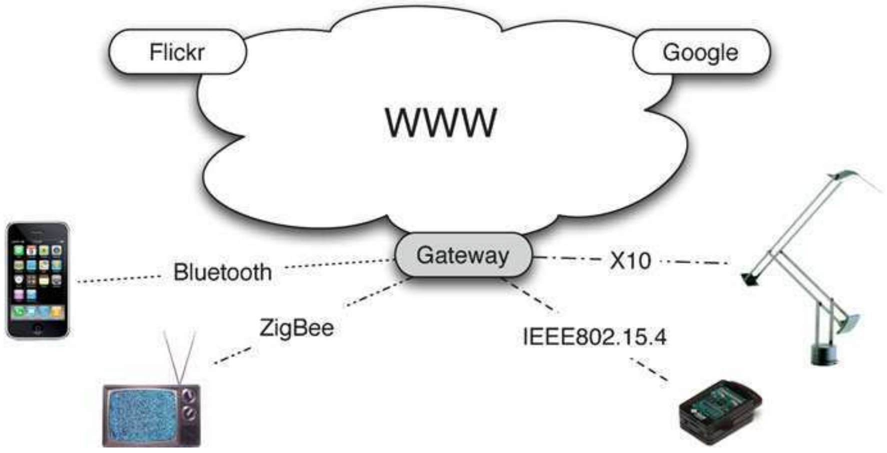


Fig. 17.1 A *gateway* is a Web application to bridge embedded devices to the Web by hiding the various low-power protocols used by devices behind a RESTful API

Hierarchical Location Modeling

A central property of the Web is the use of hyperlinks to connect related resources possibly using semantically annotated links (for example using *friend of a friend*,[3] FOAF.) To create a distributed infrastructure for smart things, we propose to bind gateways together in a similar manner. In previous work, we have explored how gateways can be linked to realize a distributed location-aware infrastructure for devices (Trifa et al. 2010). By assigning each gateway to a unique location and linking gateways together according to their spatial disposition, one can model the relations between places in the real world. In practice, this requires each gateway to maintain a list of links (URI) towards the gateways of (physically) adjacent places, and eventually to semantically annotate the nature of these links.

As illustrated in Fig. 17.2, such a Web-based hierarchical model of places enables ad hoc and mobile interaction with the real world at different levels of granularity (*country, region, city, street, building, floor, room, object*). Thanks to the layered system style of the REST architecture, each *node* (represented by a gateway) in the tree acts as an abstraction layer to interact with the devices and other gateways contained in its subtree, thus refines its parent by offering a finer granularity for clients that use the infrastructure. Such location-aware gateways are also called *location proxies*, and both terms are used interchangeably throughout this chapter. We also differentiate between two types of gateways: *virtual* and *physical*. Although identical from a software point of view, the difference lies in the fact that physical gateways (also called *terminal* gateways) must run on a computer (e.g., a wireless

[3]http://www.foaf-project.org/.

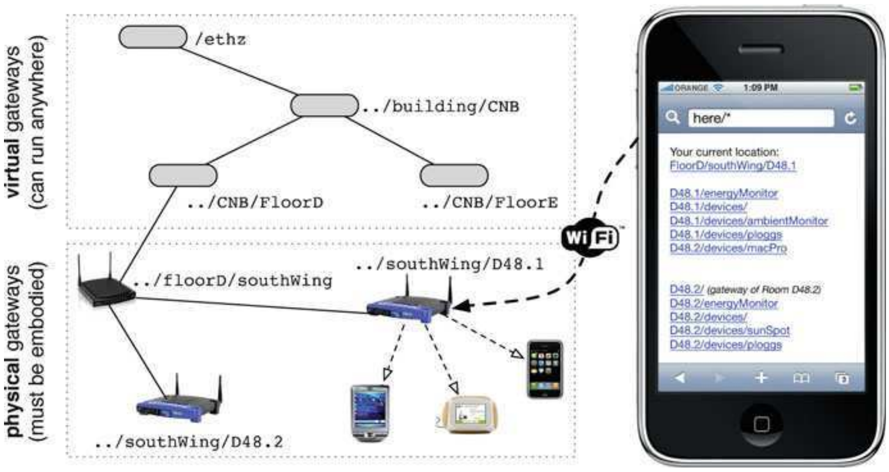


Fig. 17.2 Example gateway hierarchy from our building. The *top* gateway covers the gateways for each floor, and is composed only of virtual gateways. The `southWing` gateway runs on the router that bridges the local sub-network of that area, thus can access all terminal gateways running on computers physically located in each room. Terminal gateways have different physical interfaces to access mobile devices nearby

router, a PC, etc.) physically located in the area it maps to, and also must possess various physical interfaces to connect with devices in that location using short-range radio protocols such as Wi-Fi, ZigBee or Bluetooth. Virtual gateways, on the other hand, do not require to be installed physically located in the specific place that they act as a location proxy for, as they don't need to connect directly to physical devices. To give an example, the virtual gateways of the tree shown in Fig. 17.2 can be fully distributed across servers anywhere in the world transparently as long as the logical structure of the tree is maintained.

The mapping process that assigns the logical place name (*room 44, floor D, east wing, etc.*) to gateways must be done once manually by the developer at setup time. Fortunately, since gateways are not mobile and the structure of their connections is rather static, little effort is required to maintain the tree structure once designed. Terminal gateways can discover mobile devices in their surroundings and make them dynamically available as Web resources accessible over HTTP. This allows to navigate the tree by following links to surrounding gateways simply by clicking the links on a Web page or typing its URL in any Web browser.

On top of this network, one can easily build a system that supports range and lookup queries for mobile devices. Unlike most other hybrid models for spatial queries, our approach does not rely on a centralized database to store information about the system. Thanks to their RESTful interfaces, gateways are loosely-coupled components responsible for managing devices (and other gateways) located in the area they are associated with. The higher up a node is situated in the hierarchy, the less often things are likely to change, which naturally forms an efficient load-balancing system, as users only need to access gateways located in the area of interest without soliciting the rest of the system. As the loose coupling of the location proxies also enhances the scalability and flexibility of the infrastructure, this architecture is also particularly suited for ad hoc interaction with/from mobile devices that move across locations.

Localization

Given that many different localization techniques exist for different applications, the representation of the location information must be kept agnostic of the localization technique used to maximize flexibility and interoperability. Although many formats to represent outdoor locations have been developed recently, there is no standard way to represent indoor location information, and certainly none based on Web standards. As geographic coordinates (longitude/latitude) are not practical for dealing with location concepts used in everyday life, as for example a room's number or a building wing's name, a flexible model that supports user-generated symbolic annotations of places is needed. Sharing semantics of places can be a tedious problem in case a central authority has to maintain a repository of place names, besides it would conflict with the Web's decentralized nature.

To solve this problem, we propose to use the Web itself as a lookup service to find and explore locations, as well as to obtain information about places and the devices therein. Following the idea formulated in Jiang and Steenkiste (2002), we use URIs to represent locations and their containment relations as a logical path according to the URI definition. Consequently, RESTful URIs can be created dynamically by navigating the hierarchical tree formed by the gateways. For each URI, both machines and people should be able to retrieve a description of the identified resource. This is essential for a shared understanding about the location identified by the URI, where machines can retrieve semantically annotated data (e.g., using RDFa or Microformats) while people can retrieve a human readable representation (HTML).

Once the gateway hierarchy is in place, the problem of determining the current location of a user on the tree still remains. In particular, when several gateways are present on the same network, how does a client know which of these is the one corresponding to its location? We call this the *bootstrap problem*, and a simple method to infer the relevant location proxy's URI based on one's current location is necessary. One possibility would be to always connect automatically to the gateway with the highest signal strength, however, in practice this turns out to be very unstable as the signal strength is subject to significant and unpredictable fluctuations.

The actual spatial localization process is not part of our project, therefore we assume an indoor localization system for finding our position at the room level. For example, we could use a system such as RedPin (Bolliger 2008) to automatically return the URI of the location proxy associated with the current location. The process of binding to a gateway itself should be as easy as possible, at any place where wireless connectivity is available.

Once a physical device is associated with a gateway, its location-dependent URI can be constructed using the following syntax:

```
http://host{/location}[/keyword]
```

Here, the `host` denotes the network location of the local gateway (i.e., its IP address or network name). To traverse the location structure, `/location` is used to represent a path of arbitrary length (for example `/building44/room3/`). Finally, by specifying a `keyword`, the user can search for devices and services that match the expression. With this simple syntax, URIs become a flexible search bar. For example, to instruct a gateway to return all its links (i.e., sub-resources) to other devices or gateways, the wildcard character "`*`" can be appended to the URI of any gateway. To find all devices tagged with the keyword `phone` located on the same floor, one can simply type the following URL in any browser:

```
http://here/floor/phone/*
```

This URI can be resolved by the access point the user is associated with by using the symbolic hostname "`here/`" – such requests can always be routed to the "nearest" location gateway because the links between gateways are tagged semantically. Subsequently, a HTML page with links to all the devices that match the query and are *under* the nearest gateway named `floor` at that time will be generated dynamically. In this setting, the same URI will yield different results depending on the node in the network that it is routed to. This allows to create fixed URIs that actually point to different resources depending on the geographic location where it is issued, which is an interesting metaphor that many location-aware Web applications can benefit from.

A Distributed Modular Infrastructure for the Web of Things

A central challenge when building the Web of Things (WoT) is the development of a meaningful structure on top of individual resources attached to the WoT. Because it matches the layered architecture of the Web (Fielding 2000), we opted for the hierarchical location model described above where each node is responsible for all devices in its proximity and every proxy on a lower hierarchical level. When following this model to map physical locations to URIs, networks of gateways automatically are organized into a rooted tree, where the root represents the highest level of hierarchical location (for example the headquarters of an international organization). The hierarchical approach has been proposed in early research (Trifa et al. 2010) and shows some benefits with respect to *load balancing* and *scalability* as users mostly access devices located in their surroundings. Our efforts in designing and implementing such an architecture led towards the development of the *InfraWoT* system. An important design choice for InfraWoT was that every communication between proxies does happen locally (i.e., between neighboring nodes in the tree structure). This helps to scale the infrastructure, as each gateway only requires knowledge about its direct neighbors, thus can remain ignorant of the remaining hierarchy.

Selecting information on the hierarchical location as the main structural descriptor has immediate implications on several components of the infrastructure, for instance on the service responsible for querying within the InfraWoT tree structure or on the module in charge of maintaining the correct infrastructure internally (i.e., deciding which gateway to choose as parent and which to accept as children).

Modules Overview

As flexibility is a key requirement when implementing such an infrastructure, we decided to create location proxies that can be reconfigured without requiring a restart. To achieve this level of flexibility, we have chosen the *OSGi framework*[4] as it supports component-based development and future component-level upgrades which fosters "hot-pluggability" with other software developed for the Web of Things.

The data format used for internal information transfer is the *JavaScript Object Notation (JSON)* interchange format that provides very lightweight and easy-to-use encoding and decoding of data. Another advantage of using JSON as data format comes from the human-readable structure of this format, which greatly simplifies the debugging of software infrastructures using logs and/or live monitoring of the message streams exchanged between gateways.

The InfraWoT software consists of several modules that can interact with each other via OSGi-based messages. Each module is responsible for a specific task and implements an interface that gives access to a limited set of framework-wide functions. Figure 17.3 presents an overview of the different modules in each InfraWoT node.

- *Infrastructure Service Module.* This module maintains the correct tree structure with respect to the hierarchical locations of other proxies within its scope. As such, it takes care of child/parent registration and generates maintenance traffic between directly connected proxies (i.e., between parents and their children).
- *Discovery Service Module.* This component handles the discovery of resources, in particular the retrieval of information on resources that are to be integrated into the infrastructure and the mapping of this data to internal representations. Through this process, newly discovered resources get attached to the tree hierarchy via an InfraWoT node and thus can benefit from the services offered by the infrastructure.

[4] http://www.osgi.org.

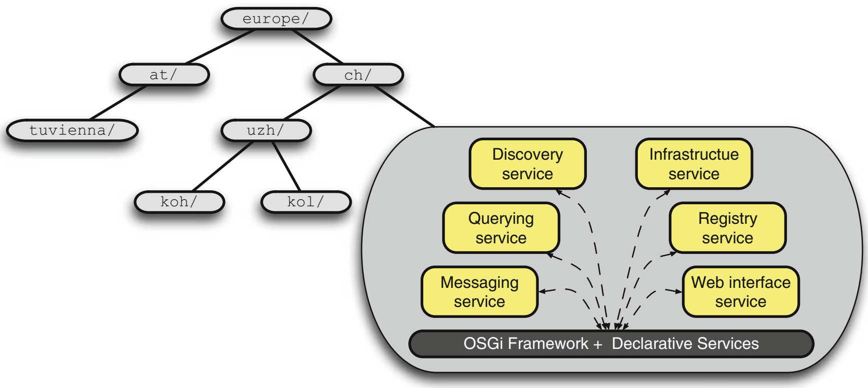


Fig. 17.3 Modules running in each InfraWoT node that interact via OSGi declarative services

- *Registry Service Module.* This component manages data about attached resources (both locally connected devices and neighbor gateways) and stores this information into a local (typically embedded) database.
- *Messaging Service Module.* This module offers a transparent interface to set up a messaging (i.e., publish/subscribe) system between client applications, gateways and physical devices attached to the Web of Things.
- *Querying Service Module.* This module is responsible for handling incoming queries. It retrieves local resources that correspond to the query and forwards the query to suitable sub- or super-nodes.
- *Web Interface Module.* This module provides a Web interface that allows to access the various functions offered by the gateway, either via a RESTful API or via an actual Web-based user interface accessible from any browser. The Web server is built upon Restlet[5] and offers various device- and gateway-specific functions.

Device and Resource Discovery Service

When a new device is connected to a network, an automated mechanism to detect the new device and to extract information about the device and how to interact with it is necessary. Many discovery protocols exist (WS-Discovery, Bonjour, etc.), however, most of them are overly complex and require an implementation of the complete discovery protocol on each device. The solution that we propose fully leverages REST to minimize the infrastructure changes required to use InfraWoT in a large-scale scenario. Furthermore, devices do not need to implement any specific

[5]http://www.restlet.org/.

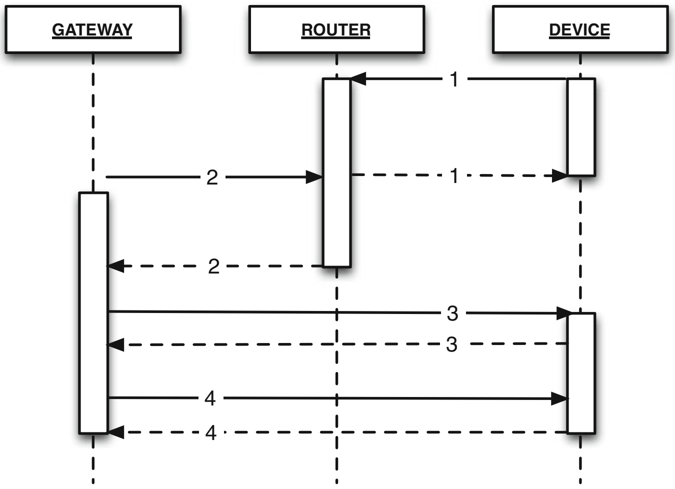


Fig. 17.4 Sequence diagram of the RESTful discovery process of devices. (1) Device connects to LAN/WiFi and gets an IP address from the router using DHCP. (2) The gateway monitors the router's DHCP table. (3) For each new device found, the gateway retrieves the root device page (by default a HTTP server running on port 80) and parses it to find information about the device. (4) The gateway retrieves the semantic description of the device

discovery protocol, but rather just have to provide semantic information about themselves in their root document. In this section, we will describe the process of attaching a new resource (i.e., a networked device featuring a RESTful interface) to the InfraWoT system.

Device Discovery

The first step of the discovery process deals with finding new WoT devices that are connected to a network. Here, we do only assume Ethernet/WiFi-enabled devices as, for other protocols, a gateway is necessary.

Most existing discovery solutions rely on devices multicasting UDP messages over the network. However, as such messages are not part of HTTP, they can often be blocked by firewalls. We therefore propose a REST-based protocol to perform network device discovery, which is shown in Fig. 17.4. We assume that in each network, the router always knows the connected network devices (usually a table of automatically assigned IP addresses), and as such can provide all required discovery information. To access this information, our solution uses *OpenWrt*[6] which is widely used, open source Linux distribution available as firmware that can run on many

[6]http://openwrt.org.

modern network routers. Its user interface – *LuCi*[7] – exposes some of its libraries and functions to external applications through a JSON-RPC API.[8]

To retrieve the list of all connected devices, an HTTP request is sent to the router (Listing 17.1):

```
Method: POST
URL: http://router/cgi-bin/luci/rpc/sys?auth=
    EBAE1814FA625E73CA0514004428D64A
Content-type: application/json
Content: {"jsonrpc": "2.0", "method": "net.arptable", "id":
    1}
```

Listing 17.1 Example of an authenticated POST command to retrieve the list of devices connected to the router from LuCi

This request will return the list of devices connected to the router by calling the RPC method `net.arptable`. Listing 17.2 shows a typical message returned by this call:

```
{"id":1,"jsonrpc":"2.0","result":[
  {"Flags":"0x2","HW type":"0x1","Device":"br-lan","Mask
      ":"*","HW address":"00:E0:4C:45:57:EF","IP address
      ":"192.168.1.114"},
  {"Flags":"0x2","HW type":"0x1","Device":"br-lan","Mask
      ":"*","HW address":"00:1C:B3:25:F6:9B","IP address
      ":"192.168.1.149"},
  {"Flags":"0x2","HW type":"0x1","Device":"eth0.1","Mask
      ":"*","HW address":"00:0D:66:22:38:01","IP address
      ":"89.211.57.1"}]}
```

Listing 17.2 Example device listing response from LuCi

The response includes a list of the IP addresses of all the physical devices connected to the router together with additional useful information. Once a list of the IP addresses of new devices that have just connected is retrieved by a proxy, the root page of each device is parsed by the Discovery Service using the procedure described in the next section (by default, the root page should be located at `http://[IP_address]:80/`).

Resource Discovery

Once a new device has been connected to the network, the second step in the discovery procedure (resource discovery) is carried out to retrieve various information about the device (functions/services, description, etc.) and make this information

[7] http://luci.subsignal.org.

[8] The current version of the LuCi is not RESTful, but as it is an open source project, the RESTful equivalent of this procedure can be easily implemented.

available within InfraWoT. In case it cannot be triggered automatically by the device discovery process described in previous section (in case the router does not offer the list of its routing table through a Web API), one needs to manually `POST` the URI of the device root page to the `/resources` endpoint of a gateway. Such requests are unpacked by the Web Interface module and the payload is relayed to the Discovery Service Bundle where the resource discovery procedure will take place.

InfraWoT provides a discovery service for Web of Things resources that is based on multiple semantic identification strategies. When a new resource is being discovered by InfraWoT, it is analyzed and mapped to an internal resource representation according to semantic markup that the resource may provide. To extract this data, InfraWoT tries to interpret any accessible representation of the resource using a number of different discovery strategies. Depending on the specific strategy, the string representing the resource is interpreted differently, for instance as a URL or as a JSON-encoded resource description. Additionally, the InfraWoT infrastructure takes into account the location information that may be provided by a discovered resource and takes care of registering that resource with the best-suited location proxy by forwarding the registration to a parent- or child-node, respectively. The different strategies have been implemented as a *Strategy design pattern* and can easily be extended, for example by implementing parsers for RDFa- or XML-based resource descriptors.

In the current version of InfraWoT, two strategies have been implemented. In the first one, InfraWoT searches the HTML resource representation found at the device URI for Microformats. Microformats provide a simple way to add semantics to Web resources. There is not one single Microformat, but rather a number of them, each one for a particular domain; a geo and adr microformat for describing places or an hProduct and hReview microformat for describing products and what people think about them. Each Microformat undergoes a standardization process that ensures its content to be widely understood and used- if accepted. More concretely, InfraWoT understands a compound of several Microformats that can be used to better describe devices. This helps for devices to be searched by humans using traditional or dedicated search engines (e.g., Google or Yahoo which are both supporting Microformats), but it also helps them being "discovered" and understood by InfraWoT in order to automatically index and use them. Currently, InfraWoT supports, but does not require, five Microformats; *hProduct* is used to describe the device itself (brand, name, picture, etc.). *hReview* reflects the quality of service or experience users or applications had with the device, *hCard* and *Geo* specify the location context of the device (address, region, country, latitude, longitude, etc.).

Finally, *hRESTS* can be used to provide additional information about the REST services that a device offers, by embedding this information directly in the devices' HTML representation. An example of the hRESTS markup to describe, for example, the *Light Sensor* resource of a sensor node is shown in Listing 17.3. It is worth noting that most of this information could be inferred by crawling the HTML representation of resources of a (truly) RESTful device and using the HTTP `OPTIONS` method. However, having this information directly embedded in the human representation of a device presents some advantages such as minimizing the HTTP calls on the device or being able to render device user interfaces in a special way, highlighting the offered services for human users. As an example, Google and Yahoo use a special HTML rendering for search results containing pages that embed Microformats such as *hReview* and *hCards*.[9]

```
<span class="hrests">
  <span class="service">
    <span class="operation">
      The
      <span class="label">Light Value</span>
      operation returning the
      <span class="output">current light value</span>
      can be invoked using a
      <span class="method">GET</span>
      at
      <span class="address">../{device}/sensors/light</span>
    </span>
  </span>
</span>
```

Listing 17.3 Microformats annotations used to describe a device and its operations, in this case a photosensor of a sensor node

The second type of discovery strategy that is currently supported by InfraWoT is based on interpreting the resource representation as a JSON object according to a pre-defined, fixed schema. While this is not realistic on an Web-wide scale, it can be used in controlled environments (e.g., in an Intranet or behind proxies such as gateways) as it is much more efficient than the Microformats-based discovery because there is no need to parse the entire device root page to find the embedded semantic annotations.

Thanks to the modular architecture of InfraWoT proxies, additional strategies can be *injected* in the Discovery Service at runtime by POSTing them to the `/strategies` endpoint of the proxy. The discovery mechanism of InfraWoT is very permissive as the minimal information necessary about a resource is the URI of that resource. If a resource provides a unique identifier within its representation, that data is incorporated as the resource's internal Unique Universal Identifier (UUID). Else, the proxy registering the object generates a new ID for unique identification of that resource. Every piece of additional information that a resource offers (e.g., using Microformats) is used to extend the resource's internal representation and thereby enables more services for that resource, for example advanced support for querying and location-aware registration.

[9]http://microformats.org/2010/07/08/microformats-org-at-5-hcards-rich-snippets.

Querying Service

Querying for resources within the scope of specific locations (such as *"find all printers in this room"*) is a central feature of any infrastructure for smart devices. InfraWoT enables such queries using various parameters such as the name of resources, their description, or the RESTful operations and parameters they accept. Additionally, InfraWoT defines several query types that encapsulate scoping information (i.e. *where* to search for resources). The handling of a search request is thus a two-step procedure that consists of first routing a query to the most appropriate gateway (e.g., the location proxy responsible for a specific building or a certain room) and second triggering it there to return the discovered resources.

A client may submit a query by sending an HTTP `POST` request to the `/query` endpoint of a proxy that contains a description of the query, either as a JSON-encoded string or using a collection of form parameters. Internally, queries are represented as JSON-serializable Java objects that contain (as mandatory parameters) an ID, the URL of the proxy that initiated the query and their type. Additionally, a query may contain an arbitrary amount of (optional) parameters that are added to the JSON representation when serialized. Such an open design facilitates upgrade and maintenance of InfraWoT (for instance queries could carry piggy-back structural information). HTTP responses to client queries can be delivered in multiple different formats, depending on the HTTP `Accept`-header specified in the request (usually JSON/XML in queries from another node/application, HTML in queries from a browser).

In principle, proxies should enable querying for all parameters that occur in the internal representation of resources. Our implementation is currently limited to those parameters that are most valuable for clients of the infrastructure. A client may search for resources using the following query types:

- *Keyword Queries.* Keyword-based search has become – thanks to the popularity of Web search engines – the most intuitive query format for many users. *Structured* queries (i.e., classical database queries) are quite complex for humans, who would rather provide textual information about the object in demand, and let the querying mechanism carry out the interpretation of this data. InfraWoT supports simple keyword-based querying by matching the provided keywords with the multiple properties and descriptions of every device in the database.
- *UUID Queries.* Particularly useful for machine–machine interaction, using the unique identifier of a device is needed when an application wants to use the infrastructure to interact with the same specific device over and over again. To humans, UUID queries are only of limited use because of the numeric format of device IDs.
- *Name Queries.* These queries enable clients to search for resources by their name and thus represent the human-useable version of UUID Queries.
- *REST Service Queries.* Matching resources according to the RESTful services they offer is a key enabler for machine–machine interaction. As the devices we have enabled for the Web of Things implement the *hRESTS* Microformat,

their HTML representation contains human-readable descriptions of their capabilities – every resource that offers services specifies its functionality with the associated *label*, HTTP *method*, *input*, *output* and *address* information, where the *input* and *output* specifiers provide a for machines to index keywords about the services of resources.

Infrastructure Service

The Infrastructure Service is used to initialize the tree structure at startup time and ensures that the correct structure is maintained during operation. In particular, this service allows the overall structure to recover from node failures and eventually re-establish the initial tree configuration (self-stabilization). After the initial setup, all gateways initialize their Infrastructure Service bundles which start the registration process with their assigned parents by sending an HTTP `POST` request that includes their own URI. Every gateway that receives such a request forwards the received URI to the Discovery Service which adds the respective gateway as a new child node.

Furthermore, the Infrastructure Service is responsible for attaching new sub-resources (i.e., other proxies or devices) found by the Discovery Service or registered manually. Any resource encountered and analyzed by the Discovery Service is passed to the Infrastructure Service which uses the resource's hierarchical location information to determine whether to attach it to the current proxy or to send it to a more appropriate gateway. In the latter case, the infrastructure takes care of routing that resource to the proxy whose hierarchical location corresponds best to the resource's (cf. Fig. 17.5). If a registering resource does not provide location information within its Web representation, the Discovery Service automatically assigns the location of the proxy itself.

Finally, it acts as *garbage collector* by regularly contacting the sub-resources and removing them from InfraWoT when they become unavailable. The Infrastructure Service starts two threads that regularly contact the parent node, all registered children nodes, and all attached resources. If the connection to any of these resources is lost, the corresponding entity gets black-listed and will be removed if contact cannot be re-established after a timeout period.

Web Interface Service

The Web Interface Service enables to access the infrastructure and the various resources connected to InfraWoT using only RESTful requests. In particular, the Web Interface Service enables the RESTful configuration of InfraWoT location proxies. We now briefly describe the individual endpoints of the InfraWoT Web interface and their functionalities:

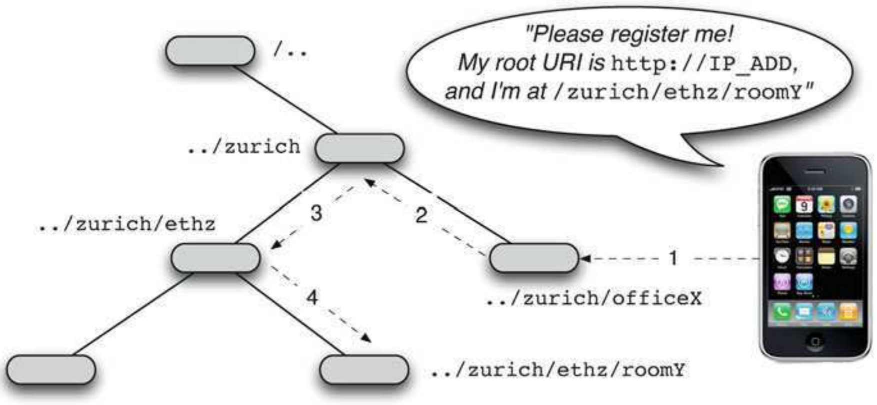


Fig. 17.5 Infrastructure-assisted discovery: any device can POST its root URI (that contains semantic information about itself) to any node in InfraWoT. If the optional location is known, the registering in routed to the node corresponding to the location specified

The *root* of any gateway ("/") provides general information on the current proxy (name, hierarchical location, connected sub-nodes, attached resources, etc.). From the root, one can access four different sub-resources (in addition to `/query` described in "Querying Service"):

The `/locations` resource represents a list of all attached location proxies. Child nodes may send HTTP `POST` requests to this address to be registered by the proxy. The HTML representation of this resource can be used to navigate (browse) the infrastructure. The individual gateways registered to any node are represented as child resources of the `/locations` resource, which can also be used to delete child nodes. For instance, to remove the gateway with UUID *ID32*, an HTTP `DELETE` request should be sent to the resource `/locations/ID32`.

The `/resources` resource represents a list of all sub-resources attached to the current gateway. Similar to the `/locations` resource, Web of Things resources may send HTTP `POST` requests to this resource to be registered by the gateway. Likewise, these resources are represented as child resources of the `/resources` resource and can be interacted with via requests to their respective endpoint within the local gateway.

The `/infrastructure` is mainly used internally by the InfraWoT software to send and receive maintenance information. One of its sub-resources, though, plays an important part in the fully Web-based configuration system of InfraWoT that enables clients to configure a proxy by sending HTTP `POST` requests to its configuration interface at `/infrastructure/configuration`. When a client `POST`s a string of data to this endpoint, the proxy relays that data to the Discovery Service to retrieve the resource encoded in the transmission and applies that information to its own representation. Although the currently preferred way to configure a gateway is to `POST` the desired configuration as a JSON-encoded resource, a gateway can be configured using any representation that is supported by the Discovery Service.

The `/messaging` resource and its sub-resources handle all interaction related to the InfraWoT Messaging Service, i.e. the creation, updating and deletion of information on the messaging interface between the gateways, client applications and physical devices.

Finally, the `/strategies` resource allows to inject additional discovery strategies at runtime (cf. "Device Discovery").

Discussion

In this chapter, we introduced InfraWoT, a flexible and scalable infrastructure for a new generation of Web applications that integrate real-time information from the physical world. As an extension of previous work with Web-enabled devices and gateways (Guinard et al. 2010; Trifa et al. 2009), InfraWoT fosters the rapid development of scalable distributed applications that incorporate data and/or functionality from heterogeneous resources on the Web of Things.

Using RESTful patterns to connect individual gateways to form a structured network that models the spatial hierarchy between places, the real-time context of Web resources (e.g., their current location) can be integrated into the Web fabric in a natural and efficient manner that allows for Web-based context-aware discovery, search and use of devices and resources. A possible scenario for this would be searching for restaurants in the vicinity according to their real-time situation (crowded, noise, etc.), by directly querying the local network infrastructure without having to fetch specific centralized Web sites.

Such a scenario could be easily implemented using InfraWoT with little or no infrastructural changes. Any existing Wi-Fi network in place would be sufficient to run InfraWoT as the discovery/query procedures fully rely on Web standards. For example, the resource discovery process we proposed can be used directly as long as a gateway and a WoT device are on the same network, by POSTing manually the root URI of the device to the gateway. This procedure could be performed automatically if routers could expose network-level information through RESTful APIs. A world where every device, gateway, or router in a network could host a local Web server offering a JSON-based API for applications and a HTML-based user interface for human users, is technically feasible today (an increasing number of routers, printers, and consumer appliances on the market today have embedded Web servers, or at least a Wi-Fi/Ethernet interface).

By describing how various functions useful for building more interactive pervasive applications can be implemented using REST, we have shown the practical advantages and flexibility offered by REST when applied to physical computing. Thanks to the layered system offered by REST, which bounds the overall system complexity and promotes the loose coupling between components, different parts of the network can be implemented independently according to the specific requirements of different applications. Using a uniform, RESTful interface for every Web of Things resource would facilitate ad hoc interaction with/between them. This way, network-level information (e.g. routing tables or network load) and real-time data from the physical world (through sensors, etc.) could be seamlessly integrated into Web applications, therefore opening a whole new range of design possibilities to make the Web *more physical* and *more real-time*.

Security and privacy issues have not been addressed in this chapter, however we are investigating the use of HTTPS and OAuth to enable authenticated and secure communication between mobile clients and gateways. A detailed performance and scalability analysis of the whole system will be necessary to better understand the dynamics and scalability of a large-scale deployment of InfraWoT.

This chapter offers a window on what the future Web might look like, and we hope to inspire Web developers to think about new possibilities that arise when combining a truly location-aware infrastructure with the Web. The work presented here shall not be taken as a finite solution, but a mere prototypical draft to foster the exploration of a future Web of Things. Much applied research and prototypes will be required before device-oriented standards for the Web become widely adopted, however, we hope our initial results and positive experiences with REST on embedded devices will stimulate further efforts to construct the Web of Things.

Acknowledgments The authors would like to thank Vlatko Davidovski for his contribution to the design and implementation of the discovery procedure.

References

Abowd, G.D., Mynatt, E.D.: Charting past, present, and future research in ubiquitous computing. ACM Trans. Comput.-Hum. Interact. **7**(1), 29–58 (2000)

Bolliger, P.: Redpin – adaptive, zero-configuration indoor localization through user collaboration. In: Workshop on Mobile Entity Localization and Tracking in GPS-less Environment Computing and Communication Systems (MELT), San Francisco (2008)

Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis, University of California, Irvine (2000)

Fielding, R.T., Taylor, R.N.: Principled design of the modern web architecture. ACM Trans. Internet Techn. **2**(2), 115–150 (2002)

Guinard, D., Trifa, V.: Towards the web of things: web mashups for embedded devices. In: Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009), in proceedings of WWW (International World Wide Web Conferences), Madrid, Spain (2009)

Guinard, D., Trifa, V., Wilde, E.: A resource oriented architecture for the web of things. In: Proceedings of IoT 2010 (IEEE International Conference on the Internet of Things). Tokyo, Japan (2010)

Jiang, C., Steenkiste, P.: A hybrid location model with a computable location identifier for ubiquitous computing. In: Proceedings of the 4th International Conference on Ubiquitous Computing, pp. 246–263. Springer-Verlag, G\&ouml;teborg, Sweden (2002). URL http://portal.acm.org/citation.cfm?id=741480

Trifa, V., Guinard, D., Bolliger, P., Wieland, S.: Design of a web-based distributed location-aware infrastructure for mobile devices. In: Proceedings of the First IEEE International Workshop on the Web of Things (WOT2010), pp. 714–719. Mannheim, Germany (2010)

Trifa, V., Wieland, S., Guinard, D., Bohnert, T.M.: Design and implementation of a gateway for web-based interaction and management of embedded devices. In: Proceedings of the 2nd International Workshop on Sensor Network Engineering (IWSNE'09). Marina del Rey, CA, USA (2009)

Wilde, E., Kofahl, M.: The locative Web. In: Proceedings of the First International Workshop on Location and the Web, pp. 1–8. ACM, Beijing, China (2008). DOI 10.1145/1367798.1367800

Chapter 18
RESTful Service Architectures for Pervasive Networking Environments

Mauro Caporuscio, Marco Funaro, and Carlo Ghezzi

Abstract Computing facilities are an essential part of the fabric of our society, and an ever-increasing number of computing devices is deployed within the environment in which we live. The vision of pervasive computing is becoming real. To exploit the opportunities offered by pervasiveness, we need to revisit the classic software development methods to meet new requirements: (1) pervasive applications should be able to dynamically configure themselves, also benefiting from third-party functionalities discovered at run time and (2) pervasive applications should be aware of, and resilient to, environmental changes. In this chapter we focus on the software architecture, with the goal of facilitating both the development and the run-time adaptation of pervasive applications. More specifically we investigate the adoption of the REST architectural style to deal with pervasive environment issues. Indeed, we believe that, although REST has been introduced by observing and analyzing the structure of the Internet, its field of applicability is not restricted to it. The chapter also illustrates a proof-of-concept example, and then discusses the advantages of choosing REST over other styles in pervasive environments.

Introduction

The Internet evolution is moving fast from "sharing" to "co-creating". The clear distinction between content producer and consumer roles, which characterized the Internet so far, is blurring towards a generic "prosumer" role that acts indistinguishably as both producer and consumer (Papadimitriou 2009). Hence, a "prosumer" is any active participant in a business, information, or social computing process. When prosumers are integrated with the computational environment and available anytime and anywhere, they are generically denoted as "things". Likewise, the term

M. Caporuscio (✉)
Politecnico di Milano, Piazza Leonardo, Da Vinci 32, 20133 Milano, Italy
e-mail: caporuscio@elet.polimi.it

Internet of Things is also often used. Due to the multitude of possible different "things" available within the environment, applications require knowledge and cognitive intelligence in order to discover, recognize, and process such a huge amount of heterogeneous information. "Things" provide services to other "things". Furthermore, it is possible to retrieve them, interact with them and change their state, and compose them to build composite "things", thus creating an *Internet of Services*.

The above concepts of prosumer, internet of things and internet of services underlie the Future Internet vision (Papadimitriou 2009), which in turn rests on the future communication and computational infrastructure. We will be virtually connected through heterogeneous means, with invisible computing devices pervading the environments (Saha and Mukherjee 2003). Such environments, referred to as *pervasive networking environments*, will be composed as spontaneous aggregation of heterogeneous and independent devices, which do not rely on predetermined networking infrastructures.

In pervasive networking environments applications emerge from compositions of the resources (the "things") available in the environment at a given time. Indeed, pervasive applications are characterized by a highly dynamic software architecture where both the resources that are part of the architecture and their interconnections may change dynamically, while applications are running. As an example, because of mobility, new things may become available dynamically, while others may suddenly disappear.

In order to face the extreme flexibility that characterizes pervasive environments, applications must support adaptive and evolutionary situation-aware behaviors. *Adaptation* refers to the ability to self-react to environmental changes to keep satisfying the requirements, whereas *evolution* refers to the ability of satisfying new or different requirements.

In order to be self-adaptable and easily evolvable, applications should exploit design models that support *loose coupling*, *flexibility*, *genericity*, and *dynamicity*. Different architectural styles and coordination mechanisms have been proposed to deal with, and reason about, distributed applications. For example, the *procedural* style, where stateless components interact via remote procedure call, or the *object oriented* style, where stateful components interact via messages. Or the *service-oriented* style, where functional or object-oriented components are not directly bound, but rather the binding may be achieved dynamically after a discovery procedure.

This chapter exploits the REpresentational State Transfer (REST) style to achieve adaptation and evolution in the context of pervasive networking environment. REST has demonstrated to be a well-suited design abstraction to deal with flexibility, genericity and dynamism (Fielding 2000), which are inherent properties of the Internet. In fact, since networked software applications are conveniently abstracted as autonomous loosely-coupled resources, they can be dynamically discovered and accessed at run time (e.g., by means of search engines), as well as combined on-the-fly to accomplish complex tasks (e.g., mashups).

The standards available for the WEB support quite effective technologies targeting the Internet domain. However, supporting WEB resource access in pervasive networking environments is still challenging. In fact, actual WEB standards essentially rely on stability assumptions associated with distributed systems and do not take into account the issues introduced by mobility (Roman et al. 2000) and, more generally, situational change, which instead permeates pervasive applications. In this case, the network structure is no longer stable and resources may come and go (physical mobility), as well as resources may move among devices (logical mobility). To comply with these constraints this chapter promotes the adoption of the REST architectural style as a design model.

The remainder of the chapter is organized as follows. "Background" describes background information on design models and software adaptation. "Why REST?" discusses why we should adopt a REST approach to address software adaptation and evolution in pervasive environments. "REST for Pervasive Systems" introduces P-REST, a meta-model for pervasive REST-oriented applications. "P-RESTful Self-adaptive Systems" illustrates how to design an adaptive and evolvable system according to P-REST. "P-REST at Work: The EXPO2015 Scenario" validates the proposed approach through a case study. "Conclusion and Final Remarks" concludes the paper and delineates future work.

Background

Research has been focusing for more than a decade on adaptive and distributed systems. Such systems have been investigated from many points of view and at different levels of abstraction. Particular attention has been devoted to architectural aspects, i.e., how to architect distributed systems to make them amenable to changes (Cheng et al. 2009). In this area, research has been mainly following two trends. On the one hand, it focused on the properties to be met by software architectures to enable applications to adapt to run-time changes. On the other, research focused on high-level models of architecture that can be kept alive at run time to support adaptation.

Since our work builds on top of both the research lines, in this section we will give a brief review of the main architectural styles emerged during the past decade and then we will go through the work on run-time models.

Architectural Styles

Many different architectural styles[1] have been proposed to deal with, and reason about, distributed systems. They can be classified according to several dimensions: (1) the type of *coupling* imposed by the model on entities; (2) the degree of *flexibility*, that is the ability of the specific model to deal with the run-time growth

[1]We also use the terms *architectural model* and *design model* interchangeably throughout the paper.

Table 18.1 Distributed design models dimensions

	Coupling	Flexibility	Genericity	Dynamism
RPC	Tight	✗	✗	✗
OO	Tight	✗	✗	✗[2]
SOA	Loose	✓	✗	✓[2]
REST	Loose	✓	✓	✗[2]

of the application in terms of involved components; (3) the degree of *genericity*, that is the ability to accommodate heterogeneous and unforeseen functionalities into the running application; (4) the kind of *dynamism*, that is the possibility to rearrange the application in terms of binding, as well as adding new functionality discovered at run time.

Table 18.1 classifies the main architectural models in terms of their characteristics with respect to the pervasive networking issues.

The oldest design model for distributed architectures is based on functional distributed components that are accessed in a synchronous way through *Remote Procedure Call* (RPC). This supports a client–server style, where: (1) client and server are tightly coupled, (2) adding/removing functions strongly affects the behavior of the overall network-based system, (3) function signatures are strict, and (4) binding between entities is generally statically defined and cannot vary (new functions cannot be discovered at run time).

Object Oriented architectures support distributed objects, and provide higher-level abstractions by grouping functions (methods) that manipulate the same object and encapsulating (and hiding) state information. The type of interaction among objects, however, is synchronous, as for the previous case. In summary: (1) interacting objects are still tightly-coupled in a client–server fashion, (2) adding/removing entities as the system is running is hard to support, (3) interfaces are specified via strict method signatures, and (4) once a reference to a remote object is set, normally it does not change at run time, and there is no predefined way of making objects discoverable (i.e., supporting this feature requires for additional ad-hoc effort).

Service Oriented Architecture (SOA) is a further step from the previous two cases because networked entities are abstracted as autonomous software services that can be accessed without knowledge of their underlying technologies. In addition, SOA opens the way to dynamic binding through dynamic discovery. In summary: (1) services are independent and loosely-coupled entities, (2) services can be easily added/removed and accessed, irrespective of their base technology, (3) service access is regulated by means of well-defined interfaces, and (4) binding between services can in principle be dynamically established at run time (although in existing SOA application this is not common practice), and new entities may be discovered and bound dynamically.

[2]This feature is conceptually feasible, although several existing instantiations of the architectural style do not support it.

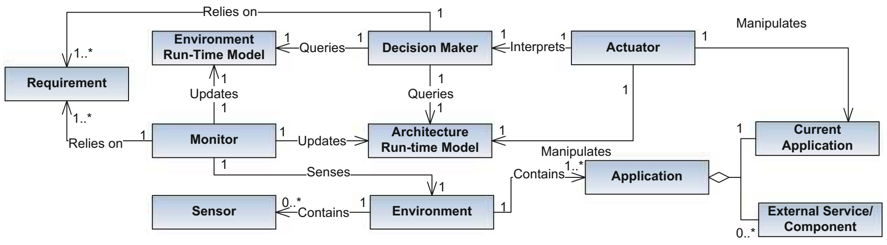


Fig. 18.1 Conceptual model for self-adaptive systems

REpresentational State Transfer (REST) differs from all the previous models in the way distributed entities are accessed and in the way their semantics is captured. REST entities are abstracted as autonomous and univocally addressable *resources*, which have a uniform interface consisting of few well-defined operations. In all previous cases, entities have different and rich interfaces, through which designers capture the semantic differences of the various entities. In REST, all entities have the same interface. Semantic information is attached separately to the identification mechanism that allows entities to be accessed. In addition, interaction with REST entities is stateless. In summary: (1) resources are independent and loosely-coupled entities, (2) resources can be easily added, removed and accessed, irrespective of their underlying technology, (3) resource access is regulated by means of a uniform interface, and (4) binding between resources is dynamically established at run time even though, in general, there is no standard way to discover and access them. However, this might be achieved by means of additional support.

Model-centric Software Adaptation

As we mentioned earlier, once an architecture is built, following some specific style, it is useful to keep a model of the architecture alive at run time to support dynamic adaptation. This section briefly elaborates on this important concept. The pivotal role played by architectural run-time models was initially recognized by Oreizy et al. (1998). In our previous work we explored this idea in two different directions in Epifani et al. (2009) and Caporuscio et al. (2010). The former paper discusses how the model can be updated as a consequence of changes observed in the environment and how this change may drive self-adaptations. The focus is on changes of the non-functional requirements of the application (performance and reliability). The latter introduces and motivates a conceptual-model (shown in Fig. 18.1) that identifies the building blocks of self-adaptive systems dealing with both *adaptation* and *evolution*. In this approach, the model kept alive at run time is composed of two sub-models, which describe the application and the environment, respectively – i.e., *Architecture Run-time Model* and the *Environment Run-time Model*.

In case of evolution, Requirements change and the Decision Maker (which in this case most likely requires human intervention) leverages the Architecture Run-time

Model to reason about the current state of the application and to devise a new abstract architecture that meets the new Requirements. The Actuator is in charge of translating the solution into an architecture and keeping the Architecture Run-time Model synchronized with the new Architecture. Adopting a suitable architectural style for describing the Architecture Run-time Model eases the decision maker's reasoning process (i.e., rules and constraints are well-known and predefined) and provides the actuator with a clear set of actions (i.e., actions are narrowed by the style's constraints) that can be performed. This also guarantees that newly devised solutions are compliant with the change by construction.

As for adaptation, an application must be aware of the environment it is working in. This is modeled by the Environment entity, which contains the applications running in an environment. An Application is described as an aggregation of the description of its architecture and of the external services or components it interacts with. The conceptual model includes the Sensors that abstract the physical context. The Decision Maker accesses the Environment Run-time Model and the Architecture Run-time Model to decide about the possible adaptive changes that need to be made to the architecture in response to changes in the environment. As opposed to evolution, adaptation is mostly achieved in a self-managed manner by the Actuator.

Why REST?

The exploitation of the REST architectural style in the context of pervasive systems is still challenging, and literature so far has been focusing mainly on interaction protocols. For example, Romero et al. (2010) exploit REST to enable interoperability among mobile devices within a pervasive environment.

On the other hand, we are interested on investigating the issues related with the design and development of RESTful applications able to evolve and adapt at run time. To this extent, this section discusses how the design of self-adaptive applications benefits from the REST principles.

The original REST architectural style (Fielding 2000) defines two main architectural entities (see Fig. 18.2): the *User Agent* that initiates a request and becomes the ultimate recipient of the response, and the *Origin Server* that holds the data of interest and responds to user agent requests. REST defines also two optional entities, namely *Proxy* and *Gateway*, which provide interface encapsulation, client-side and server-side respectively. The data of interest, held by origin servers, are referred to as *Resources* and denote any information that can be named. That is, any resource is bound to a unique *Uniform Resource Identifier* (URI) that identifies the particular resource involved in an interaction between entities. Referring to Fig. 18.2, when *User Agent* issues a request for the resource identified as R_b to *Origin Server*$_2$, it obtains as a result a *Representation* of the resource (i.e., ρ_b). Specifically, a *Representation* is not the resource itself, but captures the current state of the resource in a format matching a standard data type.

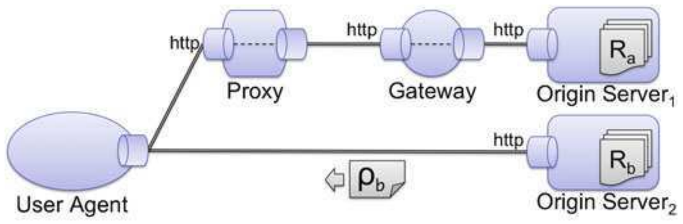


Fig. 18.2 REST architectural style

The concept of a *Resource* plays a pivotal role in the REST architectural style. In fact, it can be seen as a model of any object in the world (i.e., "things") with a clear semantics that cannot change over its lifetime. An application built according to the REST style is typically made of a set of interacting resources. An application built according to the REST architectural style is said to be "RESTful" if it does respect the four basic principles introduced by Fielding (2000) and then elaborated by Richardson and Ruby (2007): *Addressability, Statelessness, Connectedness, Uniformity*. These principles, along with the design model they induce on the application, seem to naturally apply to pervasive environments.

Addressability requires resources to have at least one URI. This RESTful applications to be found and consumed, as well as their constituent resources to be accessed and manipulated. The possibility to retrieve and use constituent resources enables prosumers to opportunistically reuse parts of a RESTful application in ways the original designer has not foreseen (Edwards et al. 2009).

The *statelessness* principle makes REST very appealing to pervasive systems. It establishes that the *state of the interaction* between a user and a RESTful application must always reside on the user side.

Since the state of the interaction is kept by the user, computations can be suspended and resumed (without losing data) at any point between the successful completion of an operation and the beginning of the next one. Indeed, using two different but equivalent resources,[3] will produce the same results. This is important in a pervasive environment since a computation, hindered by the departure of a resource, can, in principle, be resumed whenever an equivalent resource is available. Other advantages – for non-ephemeral resources – are contents "cacheability" and the possibility of load balancing through resource cloning. Hence, statelessness enhances (1) decoupling of interacting resources, (2) flexibility of the model, since it allows for easily rearranging the application at run time and, (3) scalability, by exploiting resource caching and replication. The price to pay derives from the need for an increased network capacity because the whole state of the interaction must be transferred at each request.

[3]We define two resources as equivalent iff they have the same behavior and adopt the same encoding for their representations.

The *connectedness* principle, which refers to the possibility of linking resources to one another, is the backbone of RESTful applications. It was initially introduced by Fielding in his thesis (Fielding 2000) as the "Hypermedia As The Engine Of Application State" (HATEOAS) principle. It allows for establishing dynamic and lightweight workflows such that: (1) clients are not forced to follow the whole workflow – i.e., they can stop at any time – and, (2) workflows can be entered at any time by any client provided with the proper link.

Furthermore, the state can be passed to a resource by means of the URI where it can be retrieved. In this way such a state is retrieved only if (when) needed, according to a lazy evaluation scheme.

Uniformity means that every resource must understand the core operations and must comply with their definition.

Thus, there will be no interface problems among resources. Since operations have always the same name and semantics, the genericity of the model is improved. Clearly the problem is not completely solved because data semantics and encoding must still be negotiated. It could be argued that reliance on data encoding and semantics increases the coupling between resources. However, REST eliminates the need for negotiating also the name and semantics of operations, as it happens for instance in SOA (Vinoski 2008).

Different from SOA, where service semantics is defined by means of the operations it exposes, the semantics of a resource is identified by its name. Indeed, the URI defines which semantic entity the resource models. However, as we will discuss later, this is good practice intended to ease comprehension for human beings, and cannot be applied to generic RESTful applications.

REST for Pervasive Systems

REST technologies rely on (1) the stability of the underlying communication environment and (2) tightly-coupled synchronous interaction protocols only. Pervasive environments, instead, require to (1) cope with an ever-changing communication infrastructure because devices join and leave the environment dynamically (Roman et al. 2000) and (2) to support loosely-coupled asynchronous coordination mechanisms (Huang and Garcia-Molina 2001).

This section investigates how the REST architectural style should be modified to cope with pervasive environments, and introduces the Pervasive-REST (P-REST) design model. Indeed, to make REST pervasive we need to adapt the different levels of abstraction, namely the *architecture*, the *coordination* model, and the *infrastructure*.

As we observed, in pervasive environments and, more generally, in systems envisioned for the Future Internet the role of "prosumer" will be central. Furthermore, such a prosumer role might be played by any "thing" within the environment. Hence, we foresee the necessity of departing from usual REST description of the world, made in terms of *user agents* that consume *resources* from *origin servers*

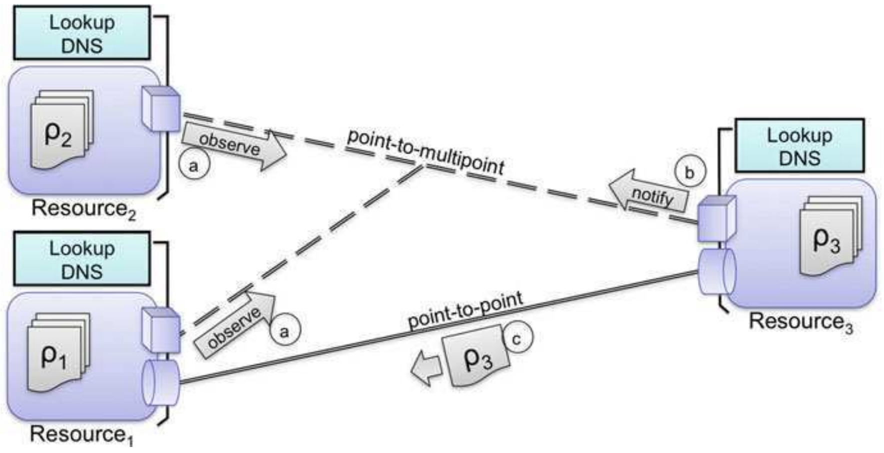


Fig. 18.3 P-REST architectural style

(see "Why REST?"). Rather, the P-REST architectural style promotes the use of *Resource* as first-class object that fulfills all roles. This means that, at the architectural level, we remove the distinction among actors, and thus we model entities in the environment as resources, which can act both as clients and servers.

To support coordination among resources, we extend the traditional request/response REST mechanism through primitives that must be supported by an underlying middleware layer. First, we assume that a *Lookup* service is provided, which enables the discovery of new resources at run time. This is needed because resources may join and leave the system dynamically. Once the resource is found, REST operations may be used to interact with it in a point-to-point fashion. The *Lookup* service can be implemented in several ways [e.g., using semantic information (Mokhtar et al. 2006), leveraging standard protocols (Romero et al. 2010)]. However, we do not rely on any specific implementation since we are focusing on the study of the design model.

The *Lookup* service yields the URI of the retrieved resource. Since resource locations may change as a result of both logical mobility (e.g., the migration of a resource from a device to another) and physical mobility (e.g., resources temporarily or permanently exiting the environment), a service is needed to maintain the maps between resource URIs and their actual location. Such service plays the role of a distributed Domain Name System (DNS) (Network Working Group 2003).

In addition, we adopt a coordination style based on the Observer pattern, as advocated in the Asynchronous-REST (A-REST) proposal described by Khare and Taylor (2004). This allows a resource to express its interest in state changes occurring in another resource by issuing an *Observe* operation. The observed resource can then *Notify* the observers when a change occurs. In this case, coordination is achieved via messages exchanged among resources.

Figure 18.3 summarizes the modification we made to the REST style. Resources directly interact with each other to exchange their representations (denoted by ρ in

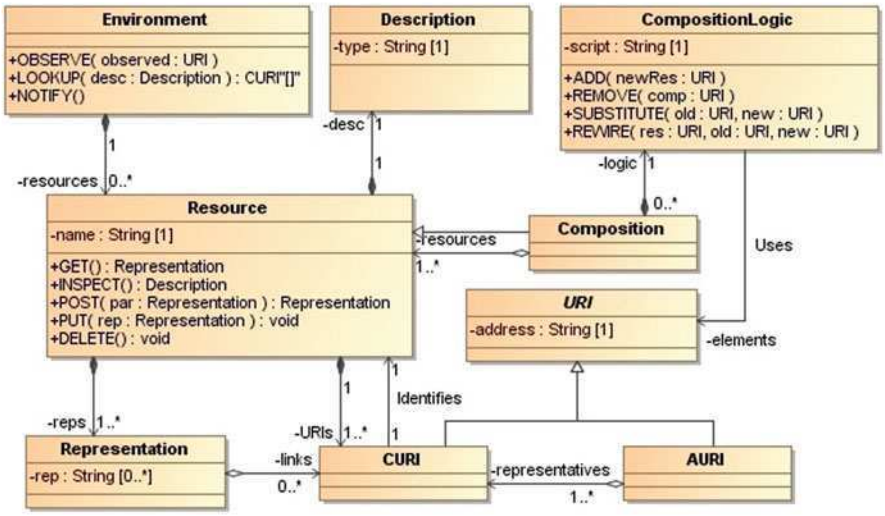


Fig. 18.4 P-REST meta-model

the figure). Referring to Fig. 18.3, both Resource$_1$ and Resource$_2$ observe Resource$_3$ (messages ⓐ). When a change occurs in Resource$_3$, it notifies (message ⓑ) the observer resources. Once received such a notification, Resource$_1$ issues a request for the Resource$_3$ and obtains as a result the representation ρ_3 (message ⓒ). Note that, while observe/notify interactions take place through the *point-to-multipoint* connector (represented as a cube), REST operations exploit *point-to-point* connector (represented as a cylinder). All the resources exploit both the `Lookup` operation to discover the needed resources, and the DNS service to translate URIs into physical addresses.

P-REST Meta-model

Along with the P-REST architectural style introduced above, we also define a P-REST meta-model (depicted in Fig. 18.4) describing the pervasive environment, the resources within the environment, and the relations among resources that define a pervasive application.

The `Environment` entity defines the whole distributed and pervasive environment as a resource container, which provides infrastructural facilities. In particular, it provides three operations that can be invoked by a resource: (1) `OBSERVE`, which declares its interest in the changes occurring in a resource identified by a given URI, (2) `NOTIFY`, which allows the observed resource to notify observers about the occurred changes, and (3) LOOKUP, which implements the distributed lookup service. These operations are the straightforward implementations of both the A-REST principle and of the lookup service, respectively.

Since Resource is a unifying first-class object, the P-REST meta-model describes every software artifact within the environment as a Resource. According to the *Uniformity* principle (see "Why REST?"), each resource implements a set of well-defined operations, namely PUT, DELETE, POST, GET, and INSPECT. The PUT operation updates the resource's internal state according to the new representation passed as parameter. The DELETE operation results in the deletion of the resource. The POST operation may be seen as a remote invocation of a function, which takes the representation enclosed in the request as input. The actual action performed by POST is determined by the resource providing it and depends on both the input representation and the resource's internal state. The semantics of the POST operation is different for different resources. This differs from the other operations whose semantics is always the same for every resource. Even if the semantics of POST is not defined by the architectural style, it is still constrained. Indeed, it can have only one semantics per-resource, and thus, overloading is not allowed. The GET implements a read-only operation that returns a representation of the resource. The INSPECT operation allows for retrieving meta-information about the resource.

REST operations can be *safe* and/or *idempotent*. An operation is considered *safe* if it does not generate side-effects on the internal state, whereas it is *idempotent* if the side-effects of $N > 0$ identical requests is the same as for a single request. GET and INSPECT operations are both idempotent and safe, PUT and DELETE operations are not safe but they are idempotent, whereas for the POST operation nothing is guaranteed for its behavior.

The REST architectural style does not provide any means to describe the semantics of resources, which is rather embedded in the URIs of resources or delegated to natural language descriptions. Instead, P-REST assumes that every resource is provided with meta-information concerning both its static and dynamic properties. As an example, for a resource representing a theater, the semantic description includes the total number of seats (a static property) as well as the number of free seats (a dynamic property). Indeed, P-REST promotes resource's semantics as first-class concern by explicitly introducing the Description entity. Specifically, Description describes both functional and non-functional properties of a resource, possibly relying on a common ontology that captures the knowledge shared by the entire pervasive environment (Berners-Lee et al. 2001). Description can also define which operations, among the available ones, are allowed or not – e.g., DELETE could be forbidden on a specific resource. Moreover, Description entities are also used to achieve dynamism (see Table 18.1). In fact, Descriptions support the implementation of the lookup service by exploiting efficient algorithms for distributed semantic discovery (e.g., Mokhtar et al. 2006), thus enabling de facto run-time resource discovery. As introduced above, Descriptions are retrieved via the INSPECT operation. Referring to the HTTP uniform interface that underlies

REST, `INSPECT` operation encapsulates both `HEAD` and `OPTION` operations and goes further by providing also the functional and non-functional specification of the target resource.

At run time, resources have their own internal state, which should be kept private and not directly accessible by other resources. The `Representation` entity overcomes such an issue by exposing a specific rendering of its internal state rather then the state itself. Hence, a Representation is a complete snapshot of the internal state, which is made available for third-party use. Every resource is associated with at least one representation, and multiple representations might be available for a given resource. This is particularly useful when dealing with heterogeneous environments in which several different data encodings are needed. A resource's representation can be retrieved by means of the `GET` operation, which can also implement a negotiation algorithm to understand which is the most suitable representation to return.

As introduced in "Why REST?", *addressability* states that every resource must be identified by means of an URI. Hence, in P-REST, every Resource is bound to at least one `Concrete URI` (cURI), and multiple cURI can refer to the same resource. Resources without any cURI are forbidden, as well as cURIs referencing multiple resources. However, P-REST enhances the concept of URI by introducing the `Abstract URI` (aURI) entity. Specifically, an aURI is a URI that identifies a group of resources. Such groups are formed by imposing constraints on resource descriptions (e.g., all the resources implementing the same functionality). The scheme used to build aURIs is completely compatible with the one used for cURI, thus they can be used interchangeably. Moreover aURIs are typically created at run time by exploiting the LOOKUP operation to find resources that must be grouped. This addition to the standard concept of URI is meant to support a wider range of communication paradigm. Indeed, using cURI allows for establishing point-to-point communication while using aURI allows for multicast communication. The latter can be useful, for instance, to retrieve the values of an entire class of sensors (e.g., humidity sensors scattered in a vineyard).

Resources can be used as building-blocks for composing complex functionalities. A `Composition` is still a resource that can, in turn, be used as a building-block by another composition. REST naturally allows for two types of compositions: *mashup* and *work-flow*. A *mashup* is a resource implemented by exploiting the functionalities provided by third-party resources. In this case, an interested client always interacts with the mashup, which in turn decomposes client's requests into sub-requests and routes them to the remote resources. Responses are then aggregated within the mashup and the result is finally returned to the client. On the other hand, a composition built as *work-flow* directly leverages the HATEOAS principle. In this case, an interested client starts interacting with the main resource and then proceeds by interacting with the resources that are discovered/created step-by-step as result of each single interaction.

Resources involved in a composition are handled by a `Composition Logic`, which is in charge of gathering resources together and, if they were not designed to interact with each other, of satisfying possible incompatibilities (e.g., handling the encoding mismatches between representations provided and expected by component resources).The composition logic is executed by a composition engine, which implements the classic architectural adaptation policies, namely *component addition*, *removal*, *substitution*, and *rewiring* (we will discuss later how such operations work). In the case of mashups, the composition logic describes how the mashup's operation are implemented; that is, how they are wired to component resources' operations. Indeed, the composition logic is the direct consequence of the exploitation of REST principles: (1) the composition is defined in terms of explicit relations between resources (i.e., connectedness), (2) resources involved in the composition are explicitly identified by means of resource identifiers (i.e., addressability) and, (3) operations on resources are expressed in terms of their interface (i.e., uniformity).

According to REST terminology, an application built following the P-REST design model is said to be P-RESTful.

P-REST Run-time Support

Traditional distributed systems differ from pervasive systems in terms of the assumptions on the underlying networking infrastructure. In particular, in pervasive systems (1) the network stability assumption is no longer guaranteed (i.e., network topology and routing strategies change over time) and (2) devices hosting resources are mobile and may have scarce processing power. Indeed, computing devices can come and go and, as a result, the network topology can change in response to either a node's arrival/departure or performance needs. Due to this new networking scenario, in order to make P-RESTful applications effective, we need to abandon the usual networking infrastructure exploited by REST. To cope with these issues, and to offer programming abstractions suitable for the rapid and efficient development of P-RESTful applications, we introduce the PRIME (P-Rest run-tIME) middleware.

Referring to Fig. 18.5, the PRIME middleware presents a layered software architecture where each layer, spanning from *transport* to *programming abstraction*, deals with specific concerns.

Transport layer: The pervasive environment, and its inherent instability calls for the adoption of a communication system resilient to structural changes (e.g, node arrival and departure). To this extent, PRIME arranges the nodes (i.e., devices) involved in the pervasive environment in a cooperative overlay network built on top of low-level wireless communication technologies (e.g., Bluetooth, Wi-Fi, Wi-Fi Direct, and UMTS). That is, each device makes use of the overlay network and, at the same time, cooperates in it by actively participating to the distributed packet routing. The transport layer is network-agnostic and does note rely on any specific technology. Indeed, it can be used on top of any IP-based network.

Coordination layer: Relying on the transport layer, PRIME provides two basic coordination mechanisms, namely point-to-point and point-to-multipoint.

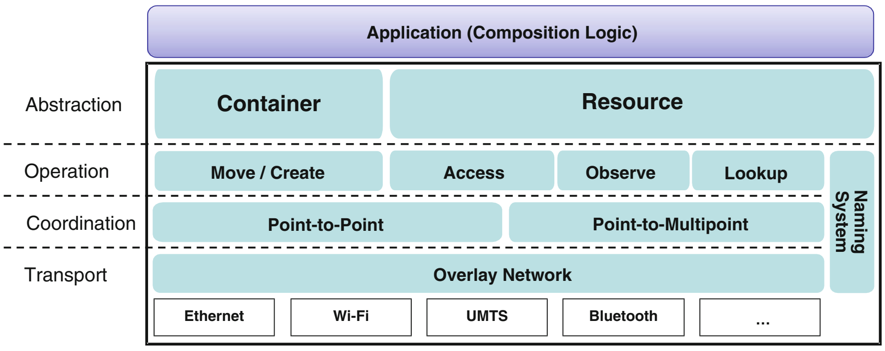


Fig. 18.5 Layered representation of PRIME

Point-to-point communication grants a given node direct access to a remote node, whereas *point-to-multipoint* communication allows a given node to interact with many different nodes at the same time.

Operation layer: The operation layer specifically deals with the concepts defined by the P-REST meta-model. In particular, it is in charge of providing the set of actions that can be performed on resources. *Access* gathers the set of operations needed to access and manipulate a resource – i.e., the set of standard REST operations provided by resources in Fig. 18.4. *Access* operations exploit the coordination layer to achieve point-to-point request-response interactions. OBSERVE allows resources to declare interest in a given resource, while NOTIFY benefits from point-to-multipoint communication and allows observed resources to advertise all the observers about occurred changes. LOOKUP allows for searching for new resources based on the description fed to it. The operation layer provides also CREATE and MOVE operations. While CREATE provides the mechanism for creating a new resource at a given location, MOVE provides the mechanism to migrate an existing resource between locations. Resource migration is useful when dealing with load balancing –by relocating the resource to an outperforming host–, device mobility –by relocating the resource to a more stable host[4]–, or energy management –by relocating a resource from a host with low battery to a lost with full battery. All the operations make use of a DNS whose task is keeping URIs consistent despite resources mobility. To this extent, the naming system shall be able to resolve URIs into physical addresses without letting resource mobility hinder such mechanism.

Abstraction layer: On top of the operation layer, PRIME provides the set of facilities and programming abstractions needed to implement P-RESTful applications. In REST, resources are held by Web servers, which handle both their life-cycle and provision. PRIME offers the same abstraction by means of *containers*. That is,

[4]Clearly, this scenario requires for additional mechanisms able to foresee whether the device leaves the environment.

each device within the pervasive environment hosts one container that, just like a Web server, handles both the life-cycle and the provision of its resources. However, unlikely Web servers, containers provide the primitives for both creating resources and migrating resources among containers (i.e., `MOVE`, `CREATE`). Their behavior, however, can be customized in order to achieve specific behaviors. For example, the `CREATE` operation can be made aware about the current load of the local container and actually allocate a resource in another similar container. As a final remark, the physical address provided by the DNS for a specific URI actually is the container's one. Indeed, a container receives all its contained resources' requests and dispatches them to the right resource based on the `C`URI.

Using the programming abstractions provided by the *Abstraction layer*, a P-RESTful application is then built as a resource that relies on other resources to meet its requirements. Specifically, the interactions between resources is specified by means of a composition language, which allows for composing and managing sets of resources. PRIME offers primitives to modify the composition logic at run tim, thus enabling architectural reconfiguration (i.e., `ADD`, `REMOVE`, `SUBSTITUTE` and `REWIRE`). We will account for these operations in "P-RESTful Self-adaptive Systems."

The PRIME APIs exploit a functional programming paradigm, which naturally achieves resource composition as the sequential application of functions. Functions are bound each other by accepting and producing immutable data structures. Immutable data structures map to resources representation, and functions are the operations exposed by resources. Through a functional language, resource compositions amounts to wiring the output of a function (i.e., operation) to the input (i.e., resource representation) of the next function. Such a functional composition can also be applied to functions that are, in turn, implemented as compositions. Thus the handling of arbitrarily complex compositions is easy and intuitive.

It is worth to note that, the abstractions provided by PRIME recall the ones introduced by CREST (Erenkrantz et al. 2007). The difference between the two approaches lies in the fact that PRIME provides such operations as infrastructural facilities, whereas in CREST resource mobility is promoted to a design principle. For such a reason we keep containers and their operations outside the P-REST meta-model. Indeed, a designer who wants to instantiate the P-REST meta-model should not be concerned with problems related to the deployment and distribution of the application.

P-RESTful Self-adaptive Systems

We argue that self-adaptive applications for pervasive systems may benefit from the adoption of the P-REST design model. To prove this, we show how the conceptual model for self-adaptive systems (Background) can be implemented by means of the P-REST meta-model (REST for Pervasive Systems), and show how the mechanisms provided by PRIME make P-RESTful application effective.

Both the conceptual model (Fig. 18.2) and the P-REST meta-model (Fig. 18.4) contain an *environment* entity. While in the conceptual model the environment is populated by generic software artifacts, in P-REST all the entities contained in the environment are modeled as a resource.

As shown in Fig. 18.2, the conceptual model revolves around the *architecture run-time model* and the *environment run-time model*. In P-REST, the architecture of the application is rendered by means of the set of resources it is composed of and the *composition logic* that orchestrates them. The type of composition used (i.e., workflow or mashup) depends on the specific functional requirements of the application. The environment run-time model is a composition of resources defined as a mashup. The corresponding composition logic is in charge of realizing the mashup by querying component resources and aggregating the results of such queries. Thus, this composition logic plays the role of the *monitor*.

Here we are not concerned with investigating how a *decision maker* might exploit the run-time models to adapt/evolve the system. Rather we want to show which mechanisms, enabled by P-REST, can be leveraged by the *actuator* to modify the running system according to decision maker's instructions. As reported by Oreizy et al. (1998), an actuator operating at the software architecture level should support two types of change: one affecting the components, namely *addition*, *removal* and *substitution*, and one affecting the connectors, namely *rewiring*.

The problem of dynamically deploying and/or removing a component from an assembly has been repeatedly tackled in literature (Kramer and Magee 1990; Vandewoude et al. 2007). Such solutions are often computationally heavy and require expensive coordination mechanisms. Moreover, preserving the whole distributed state is often very difficult since the internal state of a component is not always directly accessible. To make the problem easier, several architectural styles have been introduced. According to P-REST, adding a new resource is trivial and requires two simple steps: (1) deploy the new resource within the environment, and (2) make it visible by disseminating its URI. Once these steps are performed, the resource is immediately able to receive and process incoming requests.

On the other hand, removing a component can in general cause the loss of some part of the distributed state. P-REST, instead, works around this problem because of the stateless nature of the interactions. That is, the removed resource carries away only its internal state, thus the ongoing computations it is involved in are not jeopardized.

Substituting a component with another one cannot be simply accomplished by composing removal and addition operations. Specifically, the issue here concerns how to properly initialize the substituting component with the internal state of the substituted one. Indeed, due to information hiding it is not always possible (and not even advisable) to directly access the internal state of a component. Clearly the component can always expose part of its internal state but there is no guarantee about the completeness of the information provided. On the contrary, P-REST imposes that a resource's representation is a possible rendering of its internal state, which is always retrievable by exploiting the `GET` operation, eventhough the resource is embedded within a composition. Thus, leveraging the interaction's statelessness and the properties of a resource's representation, a P-REST resource can be substituted almost seamlessly.

As pointed out by the P-REST meta-model (see Fig. 18.2), every composition holds a composition logic describing it. Architectural run-time adaptation can be achieved by modifying the composition logic. Hence, the Composition Logic, which undertakes the run-time change, offers a specific `substitute` operation that is aware of the composing resources and of the status of requests in the composition. In particular, the semantics of the `substitute` operation is provided by means of its pseudo-code, where we leverage the PRIME container abstraction:

```
1  void substitute(cURI oldr, cURI newr) {
2      Container c = DNS.resolve(oldr)
3      c.bufferRequests(oldr);
4      c.waitForFinish(oldr);
5      Representation temp = send(GET, oldr);
6      send(PUT(temp), newr);
7      this.components.substitute(oldr,newr);
8      List<Messages> reqs = c.getPendingReqs(oldr);
9      for (Message m: reqs)
10         send(m, newr);
11 }
```

The first step of the operation retrieves the reference to the container of the old resource (i.e., the resource to be substituted). As we have already highlighted, the physical address of a container coincides with the physical address of all the resources contained in it. Thus, the `resolve` operation provided by the DNS can be exploited to retrieve, given a cURI, the physical address of the container managing the resource identified by cURI. Once retrieved, such a reference is used to access the operations offered by the container for monitoring and regulating the activities of the contained resources (i.e., their life-cycle). Line 3 instructs the container holding the old resource to buffer all the incoming requests directed to `oldr` while the substitution is taking place. As a next step, the `substitute` operation executes a blocking operation to wait for `oldr` to finish processing all the requests that are still ongoing (line 4). Now the internal state of `oldr` can be retrieved safely through its uniform interface (line 5) and used to initialize the new resource (`newr`) using a `PUT` operation (line 6). As stated above, a composition logic knows all its composing resources (through their cURIs), and we are assuming their cURIs to be stored in an instance of a data type called `components`. The instruction on line 7 substitutes the old cURI with the new one, so that the latter will always be used from now on instead of the former. Lines 8–10 retrieve the buffer of blocked requests addressed to `oldr`, and let `newr` consume them. It is important to remark that since the state of the new resource is overwritten by the `substitute` routine, it is good practice to create the new resource from scratch in order to avoid unpredictable side-effects. Indeed, if the newly inserted resource is used concurrently by other compositions, overwriting its state can be harmful. The complementary argument applies to the substituted resource. It is not deleted because it might be concurrently used by other compositions.

As for rewiring components, due to the stateless nature of the interactions, changing the URIs within the Composition Logic is sufficient for accomplishing the task. Referring to the meta-model in Fig. 18.2, the signature of the rewire operation is:

```
REWIRE (cURI res, cURI old , cURI new)
```

Its semantics is such that all the occurrences of the `old` cURI in the resource `res` will be substituted with the `new` cURI. In a mashup composition `res` is always the mashup itself because it is the only resource actually managed by the composition logic. In a workflow composition, it is important to specify `res` because it is possible that the scope of the rewiring is not extended to the whole composition, but it must be applied only to a specific point in the workflow. Unlike the `substitute` operation the state of the old resource is *not* transferred to the new one.

P-REST at Work: The EXPO2015 Scenario

In this section we describe a small case study, which is inspired by the 2015 Milan Universal Exposition (EXPO2015). We envision a city-wide pervasive environment where people, equipped with mobile devices embedding networking facilities (e.g., PDAs, smart-phones), are interconnected with each other to share information and functionalities. Any attendee may be a prosumer, acting as either participant or organizer of unexpected events.

Specifically, suppose that Carl wants to organize and promote his own BarCamp. A BarCamp[5] is an ad-hoc and spontaneous event with discussions and demos where participants, who are the main actors of the event, interact with each other sharing knowledge about a specific topic. To bootstrap his BarCamp, Carl has to (1) choose the topic and advertise the event in order to gather potential participants, (2) find and reserve a free pavilion, and (3) deploy the needed software infrastructure to handle participants' registrations.

Hereafter we address the functional design of the BarCamp application, starting from the identification of the involved resources and their relationships. Figure 18.6 sketches a simplified design of the application where some details are omitted for simplicity. We represent the Environment as an enclosing container for the resources instead of representing it as an explicit box and, as a consequence, drawing a containment relation from every other entities towards it. Also, representations and descriptions do not appear in the diagram since they are not relevant to our purpose.

[5] http://www.barcamp.org/.

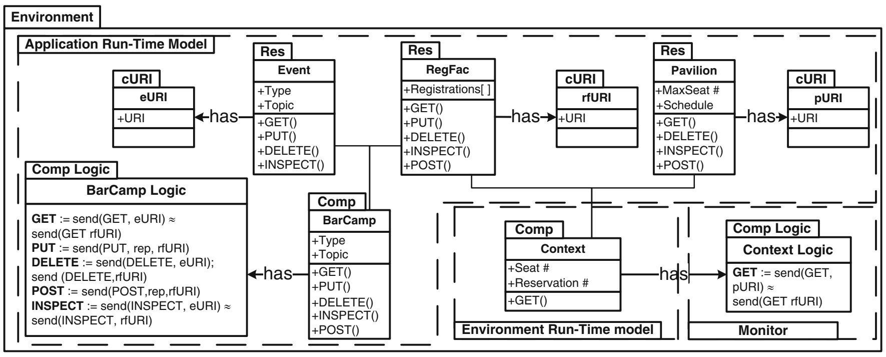


Fig. 18.6 Resource diagram of the BarCamp application

The cornerstone element of the BarCamp application described in Fig. 18.6 is the BarCamp resource, which is designed as a composition of (1) Event, which carries information about the event and (2) RegFac, which gathers attendee registrations to the event.[6] The associated composition logic, namely BarCamp Logic, defines the behavior of the operations exposed by the composite resource. The `GET` operation is computed by retrieving the current representation from both Event and RegFac and joining them (join operations are denoted by the ⊕ symbol). The actual result will be returned as a representation containing information about the event along with the registrations gathered so far. The `PUT` operation is directly mapped to the `PUT` operation provided by RegFac. The `DELETE` operation deletes the composite resource by invoking `DELETE` on Event and then on RegFac. The `POST` operation directly maps to the `POST` operation provided by RegFac. In this case, the specific semantics of POST is to create a new registration in the RegFac. The `INSPECT` operation is computed by inspecting both Event and RegFac and joining the results. The Context resource carries environmental data. It exposes only the `GET` operation that is computed by the ContextLogic by joining the number of available seats in the Pavilion and the number of registrations submitted to RegFac.

The application design, shown in Fig. 18.6, is a static description of the application and does not take into account deployment concerns, which in turn should be specified by means of different notation [e.g., UML Deployment Diagrams (Object Management Group 2010)]. Hence, we assume that resources created by Carl, namely BarCamp, BarCamp Logic, Event, Context and Context Logic will be deployed on his PDA. On the other hand, Pavilion and RegFac resources are hosted by the corresponding pavilion's infrastructural server.

According to "Background", in order to address self-adaptation the application should implement the concepts defined by the conceptual model in Fig. 18.1. The

[6]We assume that the software implementing the registration facility is provided by the Exposition Center's infrastructure as a downloadable package to foster the organization of spontaneous events.

application is implemented by BarCamp and its constituent resources (i.e., Event, RegFac and Pavilion). Hence, the architecture run-time model (dashed area in Fig. 18.6) is represented by the BarCamp composition, its constituent resources (i.e., Event, RegFac, Pavilion), and the BarCamp Logic that orchestrates the composition. The whole run-time model represents the current semantics of the application and will be the hinge of the adaptation activities.

The Context resource maps straightforwardly to environment run-time model concept, while the context logic plays the role of monitor since it is in charge of aggregating data and feeding them to the context resource (i.e., environment run-time model). The case study presented here does not use neither sensors nor external services/components. Moreover, since we are not interested in investigating solutions for automated decision-making and actuation, we assume a human-in-the-loop solution for both the Decision Maker and the Actuator roles.

Let us assume that, once advertised, the BarCamp event is very successful and the number of requests exceeds the maximum number of available seats. Carl monitors the ongoing situation by querying the context resource, and decides to adapt the application to the changing context – i.e., relocate the Barcamp to a larger pavilion. The software support for the BarCamp must adapt accordingly. The Exhibition Center's policy forbids the use of a pavilion's machinery to organizers unless they have a valid reservation for it. Since Carl is going to cancel his reservation for the first pavilion, he must substitute the original RegFac resource, which encapsulates the state of the first pavilion (i.e., the registrations gathered so far), with a new one encapsulating the new pavilion's state. Contextually, Carl does not want to restart the registration process from scratch.

To accomplish the substitution, Carl must (1) `substitute` the old RegFac resource, in both BarCamp and Context compositions, with the new one, and (2) `rewire` the old Pavilion resource with the new one in the Context composition. Note that, Pavilion is rewired, instead of being substituted, because we need to preserve the internal state of RegFac (i.e., the registrations). On the other hand, since Pavilion is a read-only resource that gathers information about the facility, it does not have an internal state to be transferred from the old instance to the new one. Thus, Carl creates the new RegFac resource and passes its URI as a parameter to the `substitute` operations exposed by BarCampLogic and ContextLogic, along with the URI of the old RegFac resource. Hence, Carl retrieves the `c`URI of the new Pavilion resource and uses it as a parameter for the `rewire` operation of the ContextLogic. In this way substitution takes place automatically.

Conclusion and Final Remarks

In this chapter we have addressed the problem of designing applications operating in evolving pervasive environments. Such applications are required to support adaptive and evolutionary situation-aware behaviors, to deal with changes occurring in the run-time environment. Changes are mainly the result of the dynamic appearance/disappearance of functionalities and the interaction with the physical context.

We presented our model-centric conceptual model, which identifies the building blocks of self-adaptive systems dealing with both adaptation and evolution. We advocated the benefits of the REST architectural style in pervasive settings (due to its loose coupling, flexibility and dynamism) and proposed Pervasive-REST (P-REST), a REST-oriented approach for designing pervasive applications. P-REST is a meta-model that can be instantiated to design applications that follow the P-REST principles. Moreover, to support the development of P-RESTful application we introduced PRIME, a distributed middleware and a development framework that both realizes the pervasive networking environment and offers programming abstractions for implementing P-REST.

Furthermore, we have shown how to render the entities of the conceptual model using the P-REST meta-model, and presented a case study for which we designed an application exploiting P-REST. Such a case study can be implemented by exploiting any of the architectural styles discussed in "Background". However, the adoption of P-REST reduces the effort of providing at design time the mechanisms needed for adaptation purposes. Indeed, by exploiting P-REST, the application can be managed at run time without the need for the designer to foresee possible adaptation issues at design time. In particular, the basic mechanisms we took advantage of are:

1. Retrieving the internal state of a component,
2. Initializing the internal state of a component,
3. "Unboxing" a composition to access one of its composing elements,
4. Run-time rebinding of components within the composition logic.

To support the same set of adaptation mechanisms within an application designed according to a traditional SOA paradigm, the designer should foresee several special cases at design time. First, the designer should figure out how to grant direct access to information embedded in a composition. Indeed, a service composition is provided (and consumed) through a set of interfaces and most of the business logic is hidden behind those interfaces. Thus, referring to the case study presented in "P-REST at Work: The EXPO2015 Scenario," an ad-hoc interface should be provided for exposing only the information regarding the registration facility. The same applies for granting access to information regarding the pavilion needed to trigger the adaptation. Finally, one more ad-hoc interface must be designed to allow for initializing the new registration facility with the old state. Moreover, dynamic binding is not directly provided by SOA, but requires for additional ad-hoc support.

We have shown instead that the adoption of P-REST allows adaptation to be carried out in a seamless way, without any special preventive actions by the designer since all the needed functionalities are imposed by the architectural style.

Acknowledgements This research has been Funded by the European Commission, Programme IDEAS-ERC, Project 227077-SMScom (SMSCom 2008).

References

Ben Mokhtar, S., Kaul, A., Georgantas, N., Issarny, V.: Efficient semantic service discovery in pervasive computing environments. Middleware 2006, pp. 240–259 (2006)

Berners-Lee, T., Hendler, J., Lassila, O.: The Semantic Web. Scientific American (2001)

Caporuscio, M., Funaro, M., Ghezzi, C.: Architectural issues of adaptive pervasive systems. In: G. Engels, C. Lewerentz, W. Schäfer, A. Schïurr, B. Westfechtel (eds.) Graph Transformations and Model Driven Enginering – Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday, *Lecture Notes in Computer Science*, vol. 5765, pp. 500–520. Springer (2010)

Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.): Software Engineering for Self-Adaptive Systems, *Lecture Notes in Computer Science*, vol. 5525. Springer, Berlin, Heidelberg, New York (2009)

Edwards, W.K., Newman, M.W., Sedivy, J.Z., Smith, T.F.: Experiences with recombinant computing: Exploring ad hoc interoperability in evolving digital networks. ACM Trans. Comput.-Hum. Interact. **16**(1), 1–44 (2009). DOI http://doi.acm.org/10.1145/1502800.1502803

Epifani, I., Ghezzi, C., Mirandola, R., Tamburrelli, G.: Model evolution by run-time parameter adaptation. In: ICSE '09, pp. 111–121. IEEE Computer Society, Washington, DC, USA (2009). DOI http://dx.doi.org/10.1109/ICSE.2009.5070513

Erenkrantz, J.R., Gorlick, M., Suryanarayana, G., Taylor, R.N.: From representations to computations: the evolution of web architectures. In: ESEC-FSE '07, pp. 255–264 (2007)

Fielding, R.T.: REST: Architectural styles and the design of network-based software architectures. Ph.D. thesis, University of California, Irvine (2000)

Huang, Y., Garcia-Molina, H.: Publish/subscribe in a mobile enviroment. In: Proceedings of the 2nd ACM International Workshop on Data Engineering for Wireless and Mobile Access, pp. 27–34 (2001)

Khare, R., Taylor, R.N.: Extending the representational state transfer (rest) architectural style for decentralized systems. In: ICSE '04, pp. 428–437. IEEE Computer Society, Washington, DC, USA (2004)

Kramer, J., Magee, J.: The evolving philosophers problem: Dynamic change management. IEEE Tran. Soft. Eng. **16**(11), 1293–1306 (1990). DOI http://dx.doi.org/10.1109/32.60317

Network Working Group: Role of the Domain Name System (DNS) (2003). RFC3467

Object Management Group: Unified Modeling Langiage Specification (2010). Version 2.3

Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based runtime software evolution. In: ICSE '98, 1998.

Papadimitriou, D.: Future internet – the cross-etp vision document. http://www.future-internet.eu (2009). Version 1.0

Richardson, L., Ruby, S.: Restful web services. O'Reilly (2007)

Roman, G.C., Picco, G.P., Murphy, A.L.: Software engineering for mobility: a roadmap. In: FOSE '00, pp. 241–258. ACM, New York, NY, USA (2000). DOI http://doi.acm.org/10.1145/336512.336567

Romero, D., Rouvoy, R., Seinturier, L., Carton, P.: Service discovery in ubiquitous feedback control loops. In: DAIS, pp. 112–125 (2010)

Saha, D., Mukherjee, A.: Pervasive computing: A paradigm for the 21st century. Computer **36**(3), 25–31 (2003). DOI http://doi.ieeecomputersociety.org/10.1109/MC.2003.1185214

SMSCom: Self Managing Situated Computing. http://www.erc-smscom.org/ (2008)

Vandewoude, Y., Ebraert, P., Berbers, Y., D'Hondt, T.: Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. IEEE Trans. Softw. Eng. **33**(12), 856–868 (2007). DOI http://dx.doi.org/10.1109/TSE.2007.70733

Vinoski, S.: Demystifying restful data coupling. IEEE Internet Computing **12**(2), 87–90 (2008)

Part VI
REST Research

Chapter 19
On Entities in the Web of Data

Michael Hausenblas

Abstract This chapter aims to explore what “entities” in the Web of Data are. As a point of departure we examine a number of widely used RESTful Web APIs in terms of URI space design and hyperlinking support in the offered resource representations. Based on the insights gained from the API review, we motivate the concept of an entity as well as its boundaries. Eventually we discuss the relevance of the entity concept for publishers and consumers of Web data, as well as the impact on Web data design issues.

The Web of Data

In this chapter, we will certainly not be able to resolve a decade-old, well-known issue from the programming domain (Post 1982):

> Nicklaus Wirth, the designer of PASCAL, was once asked, “How do you pronounce your name?”. He replied “You can either call me by name, pronouncing it *Veert*, or call me by value, *Worth*.” One can tell immediately from this comment that Nicklaus Wirth is a Quiche Eater.

However, as this chapter unfolds, we will discover that a somewhat similar problem exists concerning Web data. When we say *Web data* in the context of this chapter, we mean data as found in Web APIs and dataset collections, such as governmental statistics, eCommerce product data, or media content (such as found in Flickr), with a particular interest in widely deployed formats, including Atom, JSON, but also serialisations of the Resource Description Framework (RDF) (Klyne et al. 2004). Eventually, we are particularly interested in a characteristic of Web data that tells it apart from centralised data sources (such as relational databases): *links between data items*.

M. Hausenblas (✉)
DERI, National University of Ireland Galway, IDA Business Park, Galway, Ireland
e-mail: michael.hausenblas@deri.org

In contrast to Chap. 5, “Hypermedia Types” (by Mike Amundsen) we do not focus on the internals of the used media types or the hyperlink semantics. Also, we do not aim to address the metadata issues as discussed in Chap. 23. “Metadata Architecture in RESTful Design” (by Antonio Garrote and Maria N. Moreno Garcia), however both chapters can be seen as complimentary to the one at hand.

We will start off this chapter by reviewing existing RESTful APIs in “Reviewing RESTful APIs”, and then discuss design considerations regarding the *URI space* (URI Space Design), *representations* (On Representations and Entity Boundaries) and *hyperlinking* support (Hyperlinking) concerning entities. “Limitations and Future Work” reports on the limitations of the work presented and eventually, in “Conclusion” we conclude this chapter.

Reviewing RESTful APIs

Methodology

In the following, we will have a closer look at widely used and popular services that claim to offer RESTful APIs. The goal is to gain a deeper insight into the actual deployment status of resource granularity and the degree of hyperlinking support. In order to assess the before-mentioned aspects, we will examine the following characteristics of each API:

1. *URI space design*. In this respect, we are interested in how the API’s URI space is partitioned: we analyse if all important resources have URIs, how the URI space is organised and how cool the URIs are.[1] For example, they URI space may be organised in a flat manner or hierarchically.
2. *Representations*. Concerning the resource representations offered by the API, we ask if registered media types, such as Atom (Nottingham and Sayre 2005) are used vs. custom formats. Further, we investigate if alternative representations are offered via content negotiation or comparable mechanisms (Raman 2006).
3. *Hyperlinking*. Regarding this aspect we examine if and how hyperlinks are used. Based on the findings in the previous category we analyse the utilisation of hyperlinks in the representations served by the API. We ask especially to which extend they support the discovery of related data items within the site and potentially outside the API.

The selection of the APIs in the following is based on popularity[2] and experiences the author has gained in projects as well as from the interaction with the REST

[1] http://http://www.w3.org/Provider/Style/URI.

[2] http://http://www.programmableweb.com/apis/directory/1?protocol=REST.

community. We are well aware of the fact that the review is neither exhaustive nor definitive, nonetheless it offers an representative insight what is currently available in terms of RESTful APIs on the Web.

Each of the following sections[3] starts out with a table summarising the API characteristics regarding the above aspects and further lists more detailed observations per API. For the summary table we will use *fine-grained* if the URI space exposes all relevant resources and *coarse-grained* if only few resources are exposed. Note that our usage of fine-grained vs. coarse-grained differs from the usage typically found in the literature, where fine-grained refers to object-oriented style and coarse-grained means document-style interactions with fewer, but more structured documents.

Basecamp

Basecamp is a popular Web-based project management tool, mainly dealing with people, notes, to-do items, shared documents, milestones and time spent on activities. According to the documentation, the API[4] is "vanilla XML over HTTP using all four verbs (GET/POST/PUT/DELETE)".

URI space design	Representations	Hyperlinking
Fine-grained	Custom XML, HTML	No

1. *URI space design*. The API exposes the relevant resources of the domain in a fine-granular, hackable manner. For example, each to-do list in a project has a URI.[5]
2. *Representations*. A proprietary XML format is used as the main representation; content negotiation is supported.
3. *Hyperlinking*. There seems to be no explicit usage (or support) of hyperlinking in the XML representations.

Delicious

Delicious is a social bookmarking site, mainly dealing with people, bookmarks, and tags, with a documented API.[6]

[3]The API reviews are presented in alphabetical order.

[4]http://http://developer.37signals.com/basecamp/.

[5]http://https://lidrc.basecamphq.com/projects/4284964/todo_lists.

[6]http://http://www.delicious.com/help/api.

URI space design	Representations	Hyperlinking
Coarse-grained	Custom XML	No

1. *URI space design*. Only few of the main resources the API deals with are in fact exposed, such as posts.[7]
2. *Representations*. The API offers a custom XML without content negotiation. Interestingly, JSON format is offered for certain types of information via a separate, so called "Feed API".[8]
3. *Hyperlinking*. Although the representations contain hyperlinks (for example, in the shape of `<post href='http://example.org'>...</post>`), the data items such as people and their bookmarks are not linked.

Facebook

Facebook is a social network platform, mainly dealing with people, groups, events, messages, applications, and shared media content (images, videos, etc.) with a JSON-centric API.[9]

URI space design	Representations	Hyperlinking
Fine-grained	JSON	No

1. *URI space design*. All major resources are exposed via human-readable URIs, like people[10] or events.[11]
2. *Representations*. The main representation used in the API is JSON (no content negotiation offered) with an extension mechanism via the OpenGraph protocol[12] for integrating external content into the Facebook platform.
3. *Hyperlinking*. Hyperlinks are used in the representations, however, only for stating values such as `"link": "http://www.facebook.com/mhausenblas"` and not to relate the data items within the platform.

[7]http://https://user:passwd@api.del.icio.us/v1/posts/get.

[8]http://http://www.delicious.com/help/feeds.

[9]http://http://developers.facebook.com/docs/api.

[10]http://http://graph.facebook.com/mhausenblas/

[11]http://http://graph.facebook.com/331218348435/.

[12]http://http://developers.facebook.com/docs/opengraph

Flickr

Flickr is an image and video hosting website, mainly dealing with shared media content, people, groups, and tags with a so-called "REST API".[13]

URI space design	Representations	Hyperlinking
Fine-grained	custom XML, JSON	No

1. *URI space design*. The API exposes the main resources, such as photos[14] via distinct URIs, however the method names are explicitly encoded as URI parameters (like `?method=flickr.photos.getInfo`), which is considered a bad practice in the REST community.
2. *Representations*. Both proprietary XML and JSON are offered, albeit not via content negotiation but via a parameter (`format=json`).
3. *Hyperlinking*. One can find usages of typed hyperlinks in the served representations (such as `<url type="photopage">...</url>`, linking a photo to its page), however, in the general case the data items are connected via literal values (for example `<owner nsid="7278720@N02"... />`).

FriendFeed

FriendFeed is an aggregator service, consolidating updates from social (media) platforms, blogs, as well as news feeds with an API[15] that defaults to JSON. It mainly deals with feeds, comments, people, groups and notifications.

URI space design	Representations	Hyperlinking
Fine-grained	JSON, custom XML	No

1. *URI space design*. All important resources in the API have URIs (for example, a person's feed[16]) and follow a logical structure.
2. *Representations*. The default representation the API offers is JSON, and custom XML can be obtained (`format=json`), however not via content negotiation.
3. *Hyperlinking*. Hyperlinks are used in the provided representations, for example to represent the provenance of an entry (a link to a microblog post: `"url": "..."`), however, not to provide navigation between data items in the representations.

[13]http://http://www.flickr.com/services/api/request.rest.html.

[14]http://http://api.flickr.com/services/rest/?method=flickr.photos.getInfo&photo_id=4745449672.

[15]http://http://friendfeed.com/api/documentation.

[16]http://http://friendfeed-api.com/v2/feed/mhausenblas.

GeoNames

GeoNames is a geographical database accessible through numerous APIs,[17] mainly dealing with places, regions, weather, addresses, and geo-coordinates.

URI space design	Representations	Hyperlinking
Fine-grained	custom XML, JSON, others	No

1. *URI space design*. All main resource have dedicated URIs, such as a certain region.[18]
2. *Representations*. The API offers a wide range of representations, including the two most widely supported XML and JSON (via dedicated URIs, no content negotiation), but also other formats, such as CSV, RDF/XML, KML and RSS.
3. *Hyperlinking*. In few places one is able to spot (potentially) typed links, such as `"wikipedia":"de.wikipedia.org/wiki/Mexiko-Stadt"`, however, the relations between the resources is mainly established via literal values one can look-up accross the offered APIs.

Google Maps

The Google Maps API is really a family of APIs providing geographic data for maps applications. In the following we will focus on the Google Geocoding API[19] that converts addresses into geographic coordinates and mainly deals with addresses and geo-locations.

URI space design	Representations	Hyperlinking
Fine-grained	JSON, custom XML	No

1. *URI space design*. Each resource of interest exposed via the API has its own URI (for example, when looking up the address for a certain building[20]).
2. *Representations*. Both JSON and proprietary XML is served from respective URIs (no content negotiation).
3. *Hyperlinking*. There is no indication for hyperlinking usage in the representations.

[17] http://http://www.geonames.org/export/web-services.html.

[18] http://http://ws.geonames.org/findNearbyPlaceName?lat=53.27&lng=-9.04.

[19] http://http://code.google.com/apis/maps/documentation/geocoding/.

[20] http://http://maps.googleapis.com/maps/api/geocode/json?address=IDA+Business+Park+Galway.

Netflix

Netflix is a company that offers DVD rental and on-demand video streaming. The API deals with movies, actors, awards and the like. The Netflix OData API[21] is a representative example for Microsoft's Open Data Protocol (OData) protocol,[22] a query and access protocol building upon Atom and AtomPub.

URI space design	Representations	Hyperlinking
Fine-grained	Atom/XML, JSON	Yes

1. *URI space design*. The API exposes the main resources via distinct URIs, a certain actor,[23] for example.
2. *Representations*. Per default, the API serves Atom (Nottingham and Sayre 2005), however also offers content negotiation. For example, `curl -H "Accept: application/json"` http://odata.netflix.com/v1/Catalog/ yields the JSON representation of the catalog.
3. *Hyperlinking*. The representations partially contain typed hyperlinks as specified by the Atom standard, for example relating an actor to an award as in: `<link rel="..." href="People(189)/Awards" />`. The relations themselves[24] are not dereferencable.

Twitter

Twitter is a microblogging service allowing users to send and read other users' messages (up to 140 characters) dealing mainly with said messages, people, and geo-coordinates. The API[25] comes in a REST flavour[26] and in a so-called stream version.

URI space design	Representations	Hyperlinking
Fine-grained	custom XML, JSON, RSS, Atom	No

[21]http://http://odata.netflix.com/v1/Catalog/.

[22]http://http://www.odata.org/.

[23]http://http://odata.netflix.com/v1/Catalog/People(189).

[24]http://http://schemas.microsoft.com/ado/2007/08/dataservices/related/Awards.

[25]http://http://dev.twitter.com/doc.

[26]http://http://api.twitter.com/1/.

1. *URI space design*. All main resources are exposed via URIs (such as a certain user profile [27]) organised in a flat space.
2. *Representations*. The XML and JSON representations dominate the API and are served via dedicated URIs (`/show.json?`, for example), where for some resources (like the public timeline) also RSS and Atom formats are provided.
3. *Hyperlinking*. Although one can find hyperlinks in the representations, there is no evidence for utilising typed links to relate resources within the API.

Discussion

In terms of *URI space design*, the majority (88%) of the reviewed APIs do a great job in exposing the respective main resources in a fine-granular manner. Very often the URIs can considered to be hackable, which means a developer can easily follow a pattern in constructing them. We note that for our discussion it is of no importance if we consider the URIs opaque[28] or not; in fact hackable URIs often lead to strong coupling as the URI patterns are hard-coded for convenience reasons.

The outcome regarding the *representations* is somewhat inconclusive: some 77% serve proprietary XML, only two support established standards such as Atom, however most offer developer-friendly JSON (which seems to be sufficient for the key-value structure of most responses).

Only a single API out of nine in fact supports true *hyperlinking* in its representations. Although the URIs for the resources have typically been made available (see above), the majority of the APIs seem to ignore the potential benefits in referencing them.

URI Space Design

One of the most important – though often underrated – aspects of RESTful design is how to name the things one wishes to talk about. In more technical terms one may think of URI space design. Richardson and Ruby (2007) have documented valuable good practices regarding the URI space design, but one can also obtain helpful hints from the REST Wiki,[29] where this topic is maintained under the “Noun” label.

Whereas RESTful design in general requires to identify those things we would like to interact with, the following discussion operates under the presumption that we want to establish a fine-grained access to data items in the Web of Data.

[27]http://http://api.twitter.com/1/users/show.json?user_id=817540.

[28]http://http://rest.blueoxen.net/cgi-bin/wiki.pl?OpacityMythsDebunked.

[29]http://http://rest.blueoxen.net/cgi-bin/wiki.pl?FrontPage#nid5TL.

Naming Things

Why is it essential to name things, that is, to assign URIs to all important resources one exposes on the Web? Using a small motivation example may help shed some light on this matter. Imagine a fictitious company that wants to inform about their projects and how people are involved in it. One would expect to find, for example, the following resources: *people*, *projects*, *technologies*, *products* and respective URIs, such as:[30]

- For *collection resources*, such as all projects the company maintains, the URI might be http://company.example.com/project
- *Item resources*, for example a particular project of the company, might be identified by http://company.example.com/project/pr1 and a certain person by http://company.example.com/people/roy.est.

Now, assuming one has the URI handy, one can use the URI in an application (to obtain data from it) or link to the URI from another Web site. Obviously, if such URIs are not available, one can not achieve the above things directly, and even more seriously: the network effect is crippled.

Naming things one is dealing with on the Web, that is, minting URIs for all main resources one exposes is essential for RESTful design. Non-observance cripples the network effect.

URI Fragments for Sub-resources

A special sub-topic of URI space design worthy attention is how to deal with URI fragments[31] to identify sub-resources. Using URI fragments allows to link to things, but they do not allow for interaction with said things through the uniform interface.

Concerning the URI fragment identifiers semantics, we need to consult the *Uniform Resource Identifier (URI): Generic Syntax* (RFC3986) (Berners-Lee et al. 2005):

> The semantics of a fragment identifier are defined by the set of representations that might result from a retrieval action on the primary resource. The fragment's format and resolution is therefore dependent on the media type of a potentially retrieved representation,

[30]Note that we use the terminology for the types of resources (collection and item resources) suggested by Glenn Block in a recent blog post available via http://bit.ly/rest-resource-types.

[31]http://http://www.w3.org/DesignIssues/Fragment.html.

even though such a retrieval is only performed if the URI is dereferenced. If no such representation exists, then the semantics of the fragment are considered unknown and are effectively unconstrained. Fragment identifier semantics are independent of the URI scheme and thus cannot be redefined by scheme specifications.

We note that most media types[32] do not to specify URI fragments,[33] and where this is the case, the *Architecture of the World Wide Web, Volume One* (AWWW) (Jacobs and Walsh 2004) gives us some guidance; cf. Sects. 3.2.1 (Representation types and fragment identifier semantics) and 3.2.2 (Fragment identifiers and content negotiation) of the AWWW, especially concerning the conflict resolution mechanism for content negotiation.

Summarising, URI fragments allow to identify sub-resources in a straight-forward way. One should ensure that the fragments are made identifiable, for example in the case of HTML this would require support by document authors.[34]

However, there is a number of unresolved issues around their usage, subject to research – for example, concerning the interactions with these sub-resources[35] – and standardisation, for example regarding HTTP redirects.[36]

On Representations and Entity Boundaries

Literal-style vs. Reference-style

To understand entities in the context of the Web of Data, we will first discuss how entities can be represented, following the AWWW (Jacobs and Walsh 2004). Note that in the following, we will deliberately use the terms *representation* (Fielding et al. 1999) and *document* synonymously; later on we will go into detail regarding the notion of a document.

A fundamental characteristic of Web data is the ability to utilise "hyperlinks", essentially a URI reference between resources (Jacobs and Walsh 2004) that typically comes with some link semantics attached. To assess the type and extent of the supported link semantics concerning media types, we refer the reader to Amundsen's work on *Hypermedia Types* (Amundsen 2010).

When consuming Web data one typically has to deal with the extraction of the data structure and its values from a representation. Furthermore, data values might be provided as literal values or, through a hyperlink, as a reference to another resource. In the following we will discuss these two options (literal-style vs. reference-style) in greater detail; note, however, that this does not mean that there can not or may not exist other design options at all.

[32]http://http://www.iana.org/assignments/media-types/.

[33]With a few exceptions, such as HTML and RDF/XML.

[34]Essentially meaning that relevant elements in HTML need to be supplied with and `id` attribute.

[35]http://http://oreillynet.com/xml/blog/2008/02/addressing_fragments_in_rest_1.html.

[36]http://http://lists.w3.org/Archives/Public/www-tag/2010Oct/0003.html.

```
<div>
  <div>Name: Michael Hausenblas </div>
  <div>Residence: 32 Bushypark Lawn, Galway, Irland </div>
</div>
```

Fig. 19.1 Entity represented in HTML

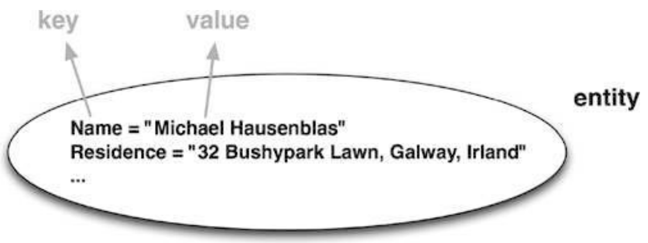

key
value
entity
Name = "Michael Hausenblas"
Residence = "32 Bushypark Lawn, Galway, Irland"
...

Fig. 19.2 Entity's key-value structure

```
<div>
 <div>Name: Michael Hausenblas </div>
 <div>Residence:<a href="address.html">my address </a></div>
</div>
```

Fig. 19.3 Entity represented in HTML

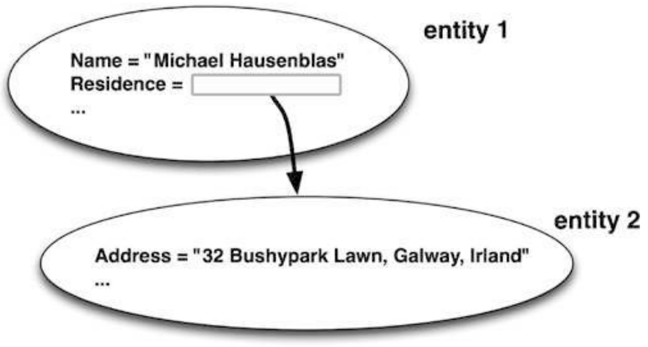

entity 1
Name = "Michael Hausenblas"
Residence =
...
entity 2
Address = "32 Bushypark Lawn, Galway, Irland"
...

Fig. 19.4 Entity's key-value structure

Providing data values via references allows for a greater flexibility compared to literal values and enables the reuse of data within a site and across the Web. However, it also comes with its costs: each reference needs to be resolved and the referred document parsed, yielding additional costs in the data processing.

Consider the following two cases: Figs. 19.1 and 19.2 show an entity with pure literal-style values, whereas Figs. 19.3 and 19.4 depict the case where partial reference-style values are used.

Although one is, in terms of hyperlinking capabilities, restricted by the choice of the representation, we note that using literal-style vs. reference-style is first and foremost a design decision of the resource owner (Jacobs and Walsh 2004). At the end of the day, one has to deal with the trade-off between processing speed (literal-style) vs. flexibility and reusability (reference-style).

Literal-style vs. reference-style data values are design decision of the resource owner, rather than characteristics of representations.

Entity Boundaries

The notion of a document has been the topic of recent discussions (cf. for example Wilde 2009). In the following, we will explore if the notion of a document is helpful in the context of the Web of Data and how this relates to entities.

To approach the issue of a "document notion", let us first step back a bit and discuss what are *directly observable things* on the Web. With directly observable we mean that something can be measured, processed, stored, etc. in the widest sense.

In a first step we want to determine what directly observable things on the Web are. As a starting point, we will use the Web's Retrieval Algorithm as described in Mendelsohn (2009): dereferencing a URI yields a representation of the resource identified by the URI. We note that *URIs* and *representations* are directly observable things, while *resources* are not directly observable. For example, a URI can be bookmarked or the representation of a resource can be stored in a file. Resources, on the other hand are purely conceptional and only are observable indirectly through URIs and the resource representation at a given point in time.

Further, we acknowledge the fact that the hyperlink structure of the Web is crucial for content discovery (Raman 2006). We note that although the discovery is enabled by hyperlinks between resources, the actual communication necessarily needs to be carried out using representations. In this context a special subset of resources is of interest: *information resource*. We will use the definition of information resource as of Jacobs and Walsh (2004), repeated here for convenience in Definition 1.

Definition 1. If all of the essential characteristics of a resource can be conveyed in a message, the resource is an information resource.

For certain applications and use cases, the concept of an information resources is essential, especially when dealing with metadata. Take for example the resource "the current temperature in Galway, Ireland". This resource is identified by the URI:

```
http://example.com/galway/temperature
```

```
<forecast city="Galway">
 <temperature>15</temperature>
</forecast>
```

Fig. 19.5 XML representation from `http://example.com/galway/temperature.xml`

```
@prefix m: <http://purl.org/ns/meteo#> .
@prefix d: <http://dbpedia.org/resource/> .
@prefix : <> .

d:Galway m:forecast :tempForecast .
:tempForecast m:temperature :tempVal .
:tempVal m:celsius "15" .
```

Fig. 19.6 RDF representation from `http://example.com/galway/temperature.ttl`

Further, assume there are two accompanying information resources:

```
http://example.com/galway/temperature.xml
http://example.com/galway/temperature.ttl
```

Having these two information resources available, one is able to state things like “the current temperature is provided to you by company X”, where it is clear that not the temperature itself, but the measurement, the data point has been made available by a certain company.

Coming back to the “notion of a document”, we now have a look at the representations (Figs. 19.5 and 19.6) retrieved from the two information resources. We understand that both convey the same information. Further, once processed by a consumer (in-memory, loaded into a relational database, etc.) one is unable to tell from which representation it originated. We, hence, claim that the notion of a document – as perceived in the XML representation – in fact has no impact on the consumer.

In fact, if one treats the resource URI and the representation together as a unit, the notion of a document as such is not helpful regarding the Web of Data. We acknowledge the fact that the above example can not be applied in a straight-forward manner to the case where the interaction with the information resource goes beyond a read-only operation (a HTTP `POST` or `PUT`, for example).

For the concept of an entity these observations are insofar essential, as the entity boundaries should not be understood in terms of document boundaries, but in terms of URIs, which potentially occur in resource representations.

Hyperlinking

Equipped with the reference-style design pattern and the idea of treating a resource URI together with its representation as a unit for processing data, we are now ready to approach the definition of an entity. While the term “entity” itself has already

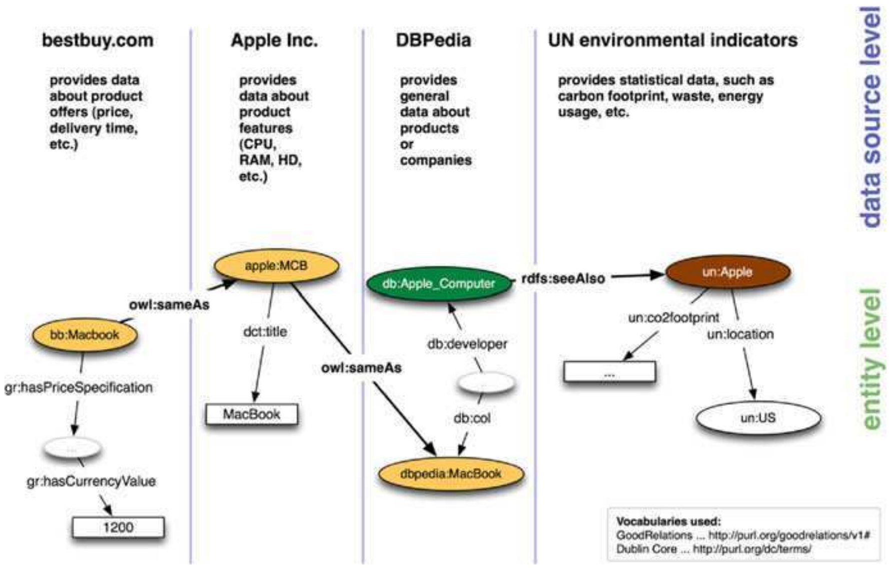

bestbuy.com
provides data about product offers (price, delivery time, etc.)
Apple Inc.
provides data about product features (CPU, RAM, HD, etc.)
DBPedia
provides general data about products or companies
UN environmental indicators
provides statistical data, such as carbon footprint, waste, energy usage, etc.
data source level
entity level
apple:MCB
owl:sameAs
bb:Macbook
gr:hasPriceSpecification
...
gr:hasCurrencyValue
1200
dct:title
MacBook
owl:sameAs
db:Apple_Computer
rdfs:seeAlso
un:Apple
db:developer
db:col
dbpedia:MacBook
un:co2footprint
un:location
...
un:US
Vocabularies used:
GoodRelations ... http://purl.org/goodrelations/v1#
Dublin Core ... http://purl.org/dc/terms/

Fig. 19.7 Entity example from the Linked Open Data realm

been in use for a while (Bouquet et al. 2008; Umbrich et al. 2010), to the best of our knowledge no agreed definition is available. We hence attempt to define an entity in the following (cf. Definition 2), and note the usage of "connector" as of Fielding and Taylor (2002).

Definition 2. An entity is a thematic view on resources across connectors, materialised through hyperlinks.

The two important bits in Definition 2 are "across connectors" and "hyperlinks"; the former acknowledging the fact that the actual data belonging to an entity is potentially distributed over several data sources and the latter that, if the data items are not explicitly connected, it is hard to impossible to construct an entity. One can even go a step further and assert that entities as such make only sense when the reference-style design is employed, as otherwise out-of-bound information is necessary to consume related information in the Web. Note also that we propose not to restrict what "thematic" might mean, as this is very likely depending on the application that processes an entity.

Take, for example, Fig. 19.7: different Linked Open Data sources[37] – homogenous linked data as of Wilde (2010) – may expose different aspects, such as price, technical features, carbon footprint, etc. regarding products through respective resources. Assume now, one is interested to buy a certain laptop with a particular price limit and carbon footprint. In this application, the entity of interest is "a laptop", and taking the interlinked data items from the four data sources together, one is able to answer the query.

[37]http://http://lod-cloud.net/.

While it seems that from the perspective of a consumer (who has to typically deal with several data sources), the concept of an entity is pretty straight-forward, in case of a data publisher it may not be so obvious. For the Web data publisher, resources and resource identifiers are the primary design elements. Not only are they (along with the choice of appropriate representations) the main building blocks, but are typically considered sufficient in terms of organisation. Regarding the data publisher side of Definition 2, with *connector* we mean in special a *server*, which is assumed to be authoritative for a resource. However, one can also understand the data publisher playing the role of a consumer regarding other data sources (Volz et al. 2009).

Limitations and Future Work

The main limitation of the research presented in this paper lies in the fact that it focuses on the read-only case (HTTP `GET`). A consistent and comprehensive discussion of the "transactional view" is subject of future research, taking into account if and how *update*, *add*, or *remove* would work without having a point of reference, that is, a container, such as a document provides.

Further, we note that it has yet to be discussed how the concept of an entity fits into proposed extensions of the REST style, such as *Computational REST* (CREST) (Erenkrantz 2009), where computational exchange is the primary exchange mechanism between peers, hence relaxing the client-server distinction found in REST.

Conclusion

The transition from document-centric processing to entity-centric processing in the Web of Data is taking place. In this chapter we have first reviewed deployed RESTful APIs in terms of their *URI space design*, concerning the served *representations* and the support of *hyperlinking* in the representations (or the lack thereof).

Based on the insights gained in the API review we have discussed challenges and pitfalls concerning the design of RESTful APIs from an entity-centric point of view, which finally leads us to the importance of the concept of an entity in the context of the Web of Data. The main idea of an entity is that it takes into account the hyperlinking aspect between Web data items and hence provides a model that goes beyond a (single) resource (from a single datasource).

Acknowledgments The author would like to thank Richard Cyganiak for the conversations regarding the notion of document, Mike Amundsen for deeper insights into hypermedia linking aspects, as well as Jonathan Rees and David Booth – in the context of the W3C task force

"Architecture of the World Wide Semantic Web" – for ongoing discussions around HTTP semantics. Last but not least, the author wants to express his gratitude to Erik Wilde for his feedback and his continuing support to ensure that the chapter focuses on under-represented topics in the REST research.

References

M. Amundsen. Hypermedia Types, 2010.

T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. Request for Comments: 3986, January 2005, IETF Network Working Group, 2005.

P. Bouquet, H. Stoermer, D. Cordioli, and G. Tummarello. An Entity Name System for Linking Semantic Web Data. In *WWW 2008 Workshop: Linked Data on the Web (LDOW2008)*, Beijing, China, 2008.

J. Erenkrantz. *Computational REST: A New Model for Decentralized, Internet-Scale Applications*. PhD thesis, University of California, Irvine, 2009.

R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. Request for Comments: 2616, June 1999, IETF Network Working Group, 1999.

R. Fielding and R. Taylor. Principled design of the modern Web architecture. *ACM Trans. Internet Technol.*, 2(2):115–150, 2002.

I. Jacobs and N. Walsh. Architecture of the World Wide Web, Volume One. W3C Recommendation 15 Dec 2004, Technical Architecture Group, 2004.

G. Klyne, J. J. Carroll, and B. McBride. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation 10 February 2004, RDF Core Working Group, 2004.

N. Mendelsohn. The Self-Describing Web. W3C TAG Finding 7 Feb 2009, Technical Architecture Group, 2009.

M. Nottingham and R. Sayre. The Atom Syndication Format. Request for Comments: 4287, December 2005, IETF Network Working Group, 2005.

E. Post. Real Programmers Don't Use PASCAL, 1982.

T. V. Raman. On Linking Alternative Representations To Enable Discovery And Publishing. W3C TAG Finding 1 November 2006, Technical Architecture Group, 2006.

L. Richardson and S. Ruby. *RESTful Web Services*. O'Reilly Media, Inc., 2007.

J. Umbrich, M. Hausenblas, A. Hogan, A. Polleres, and S. Decker. Towards Dataset Dynamics: Change Frequency of Linked Open Data Sources. In *WWW 2010 Workshop: Linked Data on the Web (LDOW2010)*, Raleigh, USA, 2010.

J. Volz, C. Bizer, M. Gaedke, and G. Kobilarov. Discovering and Maintaining Links on the Web of Data. In *ISWC '09: Proceedings of the 8th International Semantic Web Conference*, pages 650–665, Berlin, Heidelberg, 2009. Springer-Verlag.

E. Wilde. REST and RDF Granularity, 2009.

E. Wilde. Linked Data and Service Orientation. In M. Weske, J. Yang, P. Maglio, and M. Fantinato, editors, *8th International Conference on Service Oriented Computing (ICSOC 2010)*, Lecture Notes in Computer Science, page NN, San Francisco, California, 2010. Springer-Verlag.

Chapter 20
A Resource Oriented Multimedia Description Framework

Hildeberto Mendonça, Vincent Nicolas, Olga Vybornova, and Benoit Macq

Abstract This chapter presents a multimedia archiving framework to describe the content of multimedia resources. This kind of content is very rich in terms of meanings and archiving systems have to be improved to consider such richness. This framework simplifies the multimedia management in existing applications, making it accessible for non-specialized developers. This framework is fully implemented on the REST architectural style, precisely mapping the notion of resource with media artifacts, and scaling to address the growing demand for media. It offers an extensive support for segmentation and annotation to attach semantics to content, helping search mechanisms to precisely index those content. A detailed example of the framework adoption by a medical imaging application for breast cancer diagnosis is presented.

Introduction

Multimedia content have never been so widely disseminated as they are nowadays and will exponentially be from now on. This is mainly due to the simplicity of tools for the creation and dissemination of content and also to the accessibility of creative people to those tools. The production of good content is not an exclusivity of large production companies with expensive budgets anymore. Videos, music, photos, and other media are achieving multi-million audience, reveling talent artists and spreading messages that matter. However, the more content is represented as media the more it becomes difficult to be indexed by search engines and organized by applications. The drawback of this kind of content is its binary representation, whose intrinsic semantics is unclear for computers. Search engines rely on surrounding

H. Mendonça (✉)
Laboratoire de Télécommunications et Télédétection – TELE, Université catholique de Louvain, Louvain-la-Neuve, Belgium
e-mail: me@hildeberto.com

texts and few metadata to index media content, which works in most cases for search purposes but it also leads to ambiguities, misunderstandings, obscurity, besides limiting possible applications on the exploration of these data.

The identification of semantics in media content can still be done automatically by computers. There is a considerable effort from the computer science and electrical engineering communities on the recognition of patterns in images, videos, audios and so on. They are achieving impressive results in terms of precision and performance, but they are not scalable enough to describe large multimedia repositories in a reasonable time or to be integrated to a webcrawler to index media published on the web. Being more pragmatic, the description of media is more precise and detailed when done manually by users because the content might have different interpretations and might contain meanings that are not easily given artificially. Besides that, pattern recognition still fulfills an important role in the description of media due to its productivity in comparison to manual practice when making short but effective descriptions.

We can see the need for media description not only on the Web, where there is no content limitation, but also in specialized applications, where meanings are carefully inherited from one or more domains of knowledge. These kinds of applications are focused on purpose, such as video surveillance applications detecting unforeseen situations, medical imaging applications detecting intra-corporeal elements using radio and resonance, sport applications tracking players in the game to analyze moves and improve performance, and many others that generate media with meaningful content.

Once recognized, manually by human beings or automatically by computers, media content and their meanings must be directly related and stored for posterior use. In order to visualize a solution for the lack of media content's representation, as presented previously, we realized that: if media content and meanings are directly related, then they can be provided together; if provided together, applications aware of it would be able to stop relying on surrounding content and start making use of semantic descriptions. Therefore, we have elaborated a few requirements to guide the design of a solution for this purpose:

- *Define a data model to represent meanings in media content*: when meanings are ready to be associated to the content, it is important to have a data model that suits well their format and syntax, and also delimits the location where they are present.
- *Reuse what was recognized before*: keep record of recognized meanings would avoid repeating automated recognitions, saving computer resources and also helping the analysis of massive data collected throughout time. Because media content are immutable (they are not changed and if a change is needed a new version is created), none of the records becomes outdated and can be permanently reused for several purposes.
- *Assist search engines on the indexation of media content*: when storing media a minimal content description is needed. Associating meanings to content helps in the indexation process, since media can be found by textual search on all

collected meanings. At the same time, existing search engines may be used somehow as alternative ways to find content in addition to basic database querying.

- *Use open standards for representation, communication and publishing*: to make media and their meanings available for several computers, open standards should be used in order to implement full interoperability. Therefore, information stored may be represented in JSON[1] or XML[2] format, the Internet used as a communication medium, and all stored content available on demand.

With these requirements in mind, we present through this chapter a proposal to make multimedia description openly available, simplify the support for media description for developers and apply the solution to the medical imaging domain, which is relatively complex and relevant for the society. In "Multimedia Description" we describe segmentation and annotation, which are the data structure used to describe media content. We illustrate the adoption of segments and annotations in "Description of Medical Images" using a domain-specific application as an example. Then we present in "The Yasmim Framework" a framework to help developers adding support for multimedia description in existing and new applications. In the sequence, we show in "Adapting an Application to Use the Framework" how an existing application is adapted to support the framework. At last, we conclude discussing benefits and issues of this approach in "Conclusion".

Multimedia Description

A formal data model was created to support the description of multimedia files. This data structure is based on segmentation and annotation techniques. *Segmentation*, or fragmentation, consists of delimiting meaningful regions of media content (Shapiro and Stockman 2001). This process determines whether the region is useful or not for the purpose of the system application. This purpose guides the direct intervention of the end-user when manually creating segments and the implementation of recognition techniques to automatically delimit meaningful elements. *Annotation* is the assignment of meanings to segments in order to describe their content. When done manually by the user, a segment is selected and annotations are written or dragged to it. When done automatically, segments are created by recognition algorithms, indicating where the elements are located in the media, and annotated according to what the algorithm was trained to recognize.

[1] JavaScript Object Notation: http://www.json.org.

[2] eXtensible Markup Language: http://www.w3.org/standards/xml/.

Types of Segments

Spatial

The spatial segment delimits a static region in a visual media content. It can be bidimensional or tridimensional. Bidimensional segments can be used for images and video frames. Tridimensional segments can be used for 3D models. Bidimensional segments contain a set of points involving a region of interest:

$$S_s = \{(\ _1, y_1)\ \ (x_2, y_2), ..., (x_n, y_n)\} \tag{20.1}$$

where x and y are coordinates of the points in the Cartesian plane of the content. The value of x and y are correspondent to the pixels' position of a bitmap media. In a vectorial media, x and y are correspondent to the current canvas size, where the number of pixels is dynamically defined by the graphical controller. The limits for x and y are based on the image resolution and the pixels' gradient. Tridimensional segments contain a set of vertex:

$$S_s = \{(x_1, y_1, z_1), (x_2, y_2, z_2), ..., (x_n, y_n, z_n)\} \tag{20.2}$$

where x, y and z are coordinates of the vertex. The value of x, y and z are correspondent to the current canvas size, where the number of pixels is dynamically defined by the graphical controller.

The use of spatial segments is more frequent in images, whether bidimensional or tridimensional graphics, and less frequent in videos. The full content of an image can be seen at once, while a video is usually composed of thousands or even millions of frames, and its content is more meaningful when these frames are presented as a sequence. Therefore, the segmentation of only one frame might be insignificant.

Temporal

The temporal segment delimits a sequential fragment of audio or video. It defines when relevant information starts happening and when it finishes, but without any spatial delimitation. Taking an audio media as an example: a podcast is a series of digital media files that are released episodically and often downloaded through the web. These files contain rich discussions about relevant subjects, thus they are candidates to be segmented. Each interesting part of the content starts and ends in specific timestamps. The difference between the end time and start time delimits the temporal segment. It is also applicable to videos. A TV news, for instance, presents several subjects during a daily edition. These subjects can be distinguished from each other by their correspondent segments. Therefore, the definition of the temporal segment is:

$$S_t = [T_s, T_e] \tag{20.3}$$

where T_s is the starting timestamp and T_e is the ending timestamp, both included. The difference between T_e and T_s is equal to the duration of the segment.

Spatio-Temporal

The spatio-temporal segment is a merge of the spatial and temporal segments' concepts. It associates a time tag for each one of an uninterrupted sequence of frames. In an implementation level, it preserves all the characteristics of the spatial and temporal segments, but associates additional properties to the spatial segment, which are a sequential number and a correspondent time instant. Because there is a possibility of having two or more frames in the same timestamp, the sequential number was introduced to keep the sequence of those frames, independent of their timestamp. We can conclude that the formal definition of a spatio-temporal segment is:

$$S_{st} = [T_s(\{S_{s1}, S_{s2}, ..., S_{sn}\}), T_e(\{S_{s1}, S_{s2}, ..., S_{sn}\})] \tag{20.4}$$

where T is a timestamp of the temporal segment and S_s is a spatial segment in a certain instant of time. Each timestamp can be correspondent to one or more spatial segments. S_s can also represent a bidimensional (i.e. for videos and animations) or a tridimensional (i.e. for tridimensional graphics) spatial segment. This type of segment is applicable on videos, animations and 3D content and it is appropriate for tracking objects, people and other elements in the scene.

Links Between Segments

Segments by themselves are capable of meeting the needs to delimit several content samples. It allows a precise attribution of meanings to the right location, using annotations. However, there still exists gaps to fulfill in terms of representativeness. These gaps are in the limbo between segments; and our approach to fulfill them is creating *links between segments*. The arrangement of links has several configurations, such as: *sequence*, *hierarchy*, *composition*, *cause and effect*, and others. Seen as a sequence, links indicate that there is a logical order between the segments, as hierarchy they indicate the refinement of a large segment in several smaller ones, as composition they connect the parts of a bigger element, and as cause and effect they indicate the impact that a segment may cause on other segments.

Types of Annotation

The types of annotation go from a simplistic to a robust form of knowledge representation, giving more flexibility to different user profiles. They can be assigned to segments and links, covering from simple to complex media content. The supported annotations are:

Property

Property is an annotation associated to a label or a key. This key indicates the meaning of the value. Formally speaking, a property is composed of a *key* and a *value*, where the key qualifies its respective value. Properties are appropriate to describe file characteristics, for example: dimensions, resolution, size, format, volume, duration, etc. As an example of a property's syntax, we have `<size> = 25GB`, where *size* is the key and *25GB* is the value identified by the key.

Tagging

Tagging is the assignment of keywords to the media content. Each keyword represents a simple word that identifies the content in the segment or in the links between segments. Keywords are simple, efficient and widely used nowadays to create indexes of information on the web. However, it has limited representativeness when compared to other forms of annotation, although it is more practical for most people and more efficient in terms of searching because most database systems nowadays have good support for text searching.

Transcription

Transcription is a textual and complete description of a speech, dialog or music lyric. Practical applications are the automatic recognition of speech in audio sequences, sub-titles, optical character recognition (OCR) in images containing text, etc.

Description

Description is a detailed text explaining the essence of the media content. It has the same advantages and disadvantages of transcription, but with a different purpose. Practical applications are story telling material, textual summarization, situation description, scenario-based prototypes, etc.

AdHoc

AdHoc does not have commitment to be accurate in terms of content meaning. They could represent opinions, comments, external links, references, etc. AdHoc also does not have any priority in the searching mechanism and it is retrieved when the related media is already available for the user, appearing as an additional or complementary information. This is due to the fact that AdHoc annotations are informal, free-text, and can lead to erroneous decisions (Kompatsiaris and Hobson 2008).

Domain Concepts

Domain Concepts are a domain specific annotation technique. It uses ontology, which is an explicit specification of a conceptualization, providing a shared vocabulary that can be used to model concepts and their properties and relations (Gruber 1993). Concepts are more representative than tagging because they are well positioned in the domain, but they are also less efficient than tagging because there is an additional cost of exploring the graph of meanings related to them. Comparing with transcription and description, ontologies are less representative, but more explicit and computationally friendly.

The Yasmim Framework

The Yasmim Framework is an implementation of the data model used to describe multimedia content. It was designed to offer a rich description of media, attaching semantics to content of images, videos, audios, and 3D models. Using Yasmim, developers and researchers do not deal with the usual complexity of managing media content. An API to perform operations over those media is made available through web services. Therefore, instead of developing one more multimedia archiving system, we are contributing by simplifying the way multimedia is added to existing applications and making distributed multimedia management accessible for non-specialized developers.

The designed architecture aims to provide scalability, extensibility, and robustness. It is scalable because it is stateless (i.e. it does not save any state or temporary data, such as user's sessions, navigation, etc.) and uses several databases to support different kinds of data. It is extensible because several existing solutions can be used with a minimal integration effort. It is robust because the chosen technologies have been extensively applied on many other solutions, with years of experience and large communities around them.

Yasmim runs entirely on the server. Its user interface is essentially administrative. In order to access and maintain media, application clients should be developed to access the server. The communication is made through web services, using Internet protocols. They are also stateless, thus any temporal data should be managed by the client and sent to the server when necessary. Being stateless allows the system to dedicate all its computational resources to process media.

Figure 20.1 depicts the general architecture of a system using Yasmim for multimedia archiving. Yasmim is right in the middle, intermediating data between

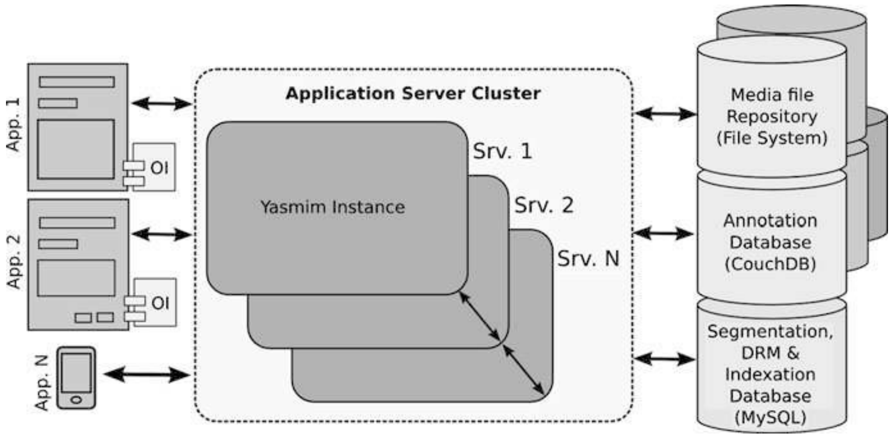

App. 1
OI
App. 2
OI
App. N
Application Server Cluster
Yasmim Instance
Srv. 1
Srv. 2
Srv. N
Media file Repository (File System)
Annotation Database (CouchDB)
Segmentation, DRM & Indexation Database (MySQL)

Fig. 20.1 General architecture

several data sources and several clients. In theory, the middleware behind it can be any application server available on the market that implements recent Java Enterprise Edition (Java EE) Specification Requests (Java EE JSR).[3] However, we have tested it only with the Glassfish.[4] It can manage instances of the same application on spread machines, expanding the processing capability according to users' demands.

The application manages the information that come from clients and organizes them in different databases. Each database was chosen according to the data that they were designed to store. These databases are:

- *Media File Repository*: Media files are stored directly in the file/storage system. The optimal efficiency on file access depends on the operating system and the storage system in use. Files are located by name, which is not exactly the original name, but the resource id registered in the database. To verify consistency, a batch process checks periodically whether there is a database record for each stored file. Orphan files are deleted in this process.
- *Segmentation and Indexation Database*: A relational database is used to store references to files in the repository because of its robust indexation mechanisms. It is also used to save segmentation data because tables have better support to store and retrieve numbers, since segments are basically coordinates and/or timestamps.
- *Annotation Database*: Annotations are stored in a document database system, which processes text more efficiently than relational databases.

[3]Java Community Process Java EE: http://jcp.org/en/jsr/platform?listBy=3&listByType=platform.
[4]Glassfish Application Server: https://glassfish.dev.java.net/.

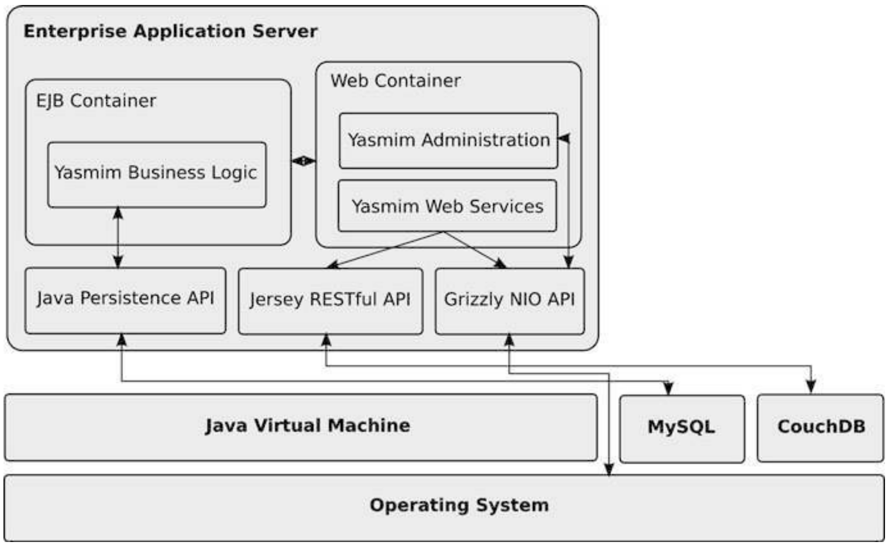


Fig. 20.2 Yasmim software architecture

The role of the client side is to process heavy operations, such as the support for several modalities, automatic segmentation, automatic extraction of meanings, and also to provide rich user interaction for intuitive manual segmentation and annotation. The data is synchronized with the server, making the media and all related data available for searching and sharing.

The server side provides REST web services (Fielding 2000), which is compatible with several kinds of clients developed in different languages, platforms and devices. According to REST architectural principles, the main data abstraction is a resource, which is represented by a media resource in our architecture. Every media and related information are reachable through unique identifiers, following the principle of addressability. Identifiers are known as URI (Uniform Resource Identifier),[5] which is used by the HTTP protocol to locate resources on the web (Richardson and Ruby 2007). With a REST-based framework, we could attach segments and annotations to media, making slight modifications on the URI. This way, not only search mechanisms can benefit from the media description, but many other practical applications as well, since REST web services are easily accessible by any socket library.

Figure 20.2 shows the server side where Yasmim runs. The application server has two execution environments: The EJB Container and the Web Container. The first one is appropriate to handle transactional data, which is suitable for operations with the relational databases. The second one is appropriate to handle and generate

[5]http://www.w3.org/TR/uri-clarification/.

content based on the target users. This content is stored and retrieved by the EJB Container and other sources and appropriately transformed for different purposes. The EJB Container runs Yasmim's business logic that takes care of the database data. The Web Container runs Yasmim's administration, which is the user interface to perform back office (maintenance) operations, and the web services, which are the interface with clients.

These containers have some facilities to access data. The *Java™Persistence API*[6] is a Java™specification for relational data access. It is capable of mapping table with Java™classes in order to transform table tuples in Java™objects, consequently simplifying data access. The *Jersey RESTful API*[7] is an implementation of the Java™specification for providing and consuming REST web services. It is used to implement and to consume REST web services. The *Grizzly NIO API*[8] is an implementation of the New Java™IO specification. It is used to store, retrieve and modify media files asynchronously, optimizing parallelism.

The platform is composed of the Java™Virtual Machine (JVM), MySQL database, and CouchDB document database. They run on top of the operating system, which is also responsible for the media storage. The JVM is responsible for the execution of Java™programs, which includes the application server and the applications running on it. MySQL[9] is one of the fastest relational databases available and its indexation, relational and transactional features are essential to deal with a large amount of numbers and unique references, which should be consistent. Finally, because annotations perform an important role in this research, they should be stored in a high scalable way, such as the one provided by CouchDB,[10] a document based database (Anderson et al. 2010). All data in CouchDB is accessible by REST web services, allowing clients to access it directly, without Yasmim mediation, although only Yasmim can write data there.

Catalog of Services

The relevant services for general understanding are listed on Table 20.1. The first column indicates the name of the service, helping the developer to identify which service is more appropriate for his/her needs. Second column shows the HTTP methods, that could be GET, POST, PUT, and DELETE. The third column shows the relative URI, starting with "http://[server-name/domain]/resources". The brackets indicate that there is a value to fulfill. This value could be pre-defined, which is the case of [type], or generated, which is the case of [id]. The last column lists the

[6]JSR 317:http://jcp.org/en/jsr/summary?id=317.

[7]JSR311:http://jcp.org/en/jsr/summary?id=311.

[8]JSR51:http://jcp.org/en/jsr/detail?id=51.

[9]http://www.mysql.com.

[10]http://couchdb.apache.org.

Table 20.1 Catalog of RESTful web services

Service	Method	URI	Parameters
Save media	POST	.../[type]s	
Get media	GET	.../[type]s/[id]	version=#
			width=#
			height=#
			rotate=#
			filter=*filter-name*
			sample=*true/false*
		.../[type]s	search=*keywords*
Remove media	DELETE	.../[id]	version=#
Save segment	POST	.../[media-id]/segments	
Get segments	GET	.../[media-id]/segments	shape=*shp-name*
			type=*type-name*
			duration=#
			search=*keywords*
Get segment	GET	.../[media-id]/segments/[id]	binary=*true/false*
Update segment	PUT	.../[media-id]/segments/[id]	
Remove segment	DELETE	.../[media-id]/segments/[id]	
Save annotation	POST	.../segments/[seg-id]/annotations	
Get annotations	GET	.../segments/[seg-id]/annotations	search=*keywords*
			type=*type-name*
		.../[media-id]/annotations	search=*keywords*
			type=*type-name*
Get annotation	GET	../segments/[seg-id]/annotations/[id]	
Update annotation	PUT	../segments/[seg-id]/annotations/[id]	
Remove annotation	DELETE	../segments/[seg-id]/annotations/[id]	

parameters to be appended to the URI. None of the parameters is mandatory, except for the parameter "search" in the "Get Media" service to avoid a high amount of records retrieved.

The value [type] can assume the following values: "image", "video", "audio", and "3d", which are the types of media supported by Yasmim. These values are mainly useful to summarize the available services. [id] is a UUID,[11] an alphanumeric string of 32 characteres with so many combinations that it theoretically never repeat for two different records. Because each id is unique, data synchronization, replication and merges are very simplified. UUID is used to define all ids, thus [id], [media-id], and [seg-id] follow the same rules.

Each parameter starts after a semicolon and can be written in any order. Some parameters are not appropriate for all types of media. "rotate", for example, cannot be applied to an audio file (Get Media) and "duration" cannot be applied to a spatial segment (Get Segments).

[11]Universally Unique Identifier: http://www.ossp.org/pkg/lib/uuid/.

Only media cannot be updated because media files are immutable. Segments and annotations can be inserted, updated, queried and deleted normally. In case a media file needs to be updated, a new version is created with the new file and the previous version is kept historically. The implementation of filters on the server would impact the overall performance. However, the decision to implement them was made because they are atomic operations, which means that there is only one algorithm for each filter and its output is exclusively used by the client. In order to improve performance, we save a version of the filtered image to retrieve in case the it is requested once again in the future, working as a buffer. The same rules are valid for format. The saved versions have a different file name pattern. Besides the id, the name also have a sequential number and the retrieval of the correct file is managed by the framework.

The hypermedia aspect of the services helps to retrieve media resources according to their respective mime types and also informs to application clients the available filters for each requested kind of media. Yasmim does not offer other references beyond these ones, thus hypermedia is not seen as a workflow but a set of options available for retrieving and processing resources.

Description of Medical Images

Annotation of medical images consists of segmenting and annotating relevant elements in images produced by hospital equipments, such as radiography, ultrasound, magnetic resonance and others. Taking breast radiography as an example, the image may depict anomalies in the breast region that might be a tumor. The analysis of a specialist (doctor) will determine whether the anomaly is a tumor, a calcification, or any other possible diagnosis.

MedicalStudio

There are applications to help doctors on the analysis of such media content. The one that we are taking into consideration is MedicalStudio because we have access to the source code and the application needs a rich support for segmentation and annotation of medical images. MedicalStudio is a component-oriented platform designed to ease the creation of medical imaging workstations (Trevisan et al. 2007). Besides simplifying the work of developers, MedicalStudio also streamlines clinical trials and end-user's experience. The platform provides a collection of reusable components that can be assembled to produce new applications considering image processing, data access, interaction design and others. Each assembly of components will produce a different application that may target different uses. In the case of an image registration application, for example, each algorithm will be seen as a component, and there will be several UI components to meet different user

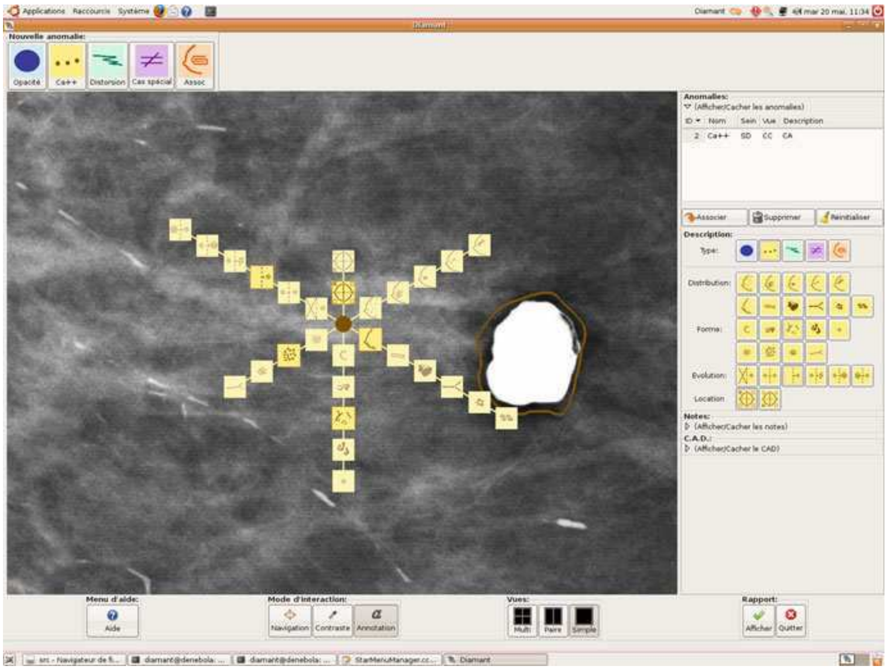


Fig. 20.3 MedicalStudio running components for mammography

profiles: (a) a configuration UI for tuning algorithms used by engineers; (b) another configuration UI for tuning options oriented for clinical researchers; and (c) a visualization UI for doctors to perform their specific clinical diagnosis.

Figure 20.3 depicts MedicalStudio's user interface, where an image of a mammography is shown. Looking at the image, the doctor can visually identify micro-calcifications, select them using spatial segments and annotate these segments using domain-specific annotations, which is well known by the doctor, who is a specialist in the field.

The platform is entirely written in C++ and relies on well accepted and powerful libraries, such as Visualization Toolkit (VTK[12]) for visualization, Insight Segmentation and Registration Toolkit (ITK[13]) for image segmentation and registration, DCMTK[14] for Digital Imaging and Communications in Medicine (DICOM[15]) interoperability and GTKmm[16] for graphical user interface. These libraries are not

[12]http://www.vtk.org.

[13]http://www.itk.org.

[14]http://dicom.offis.de/dcmtk.php.en.

[15]http://dicom.offis.de.

[16]http://www.gtkmm.org.

always enough in specific cases, so that list is not fixed and the architecture is flexible enough to allow interoperability with any other toolkit as far as it can be bound with C++.

Breast Diagnosis Domain Representation

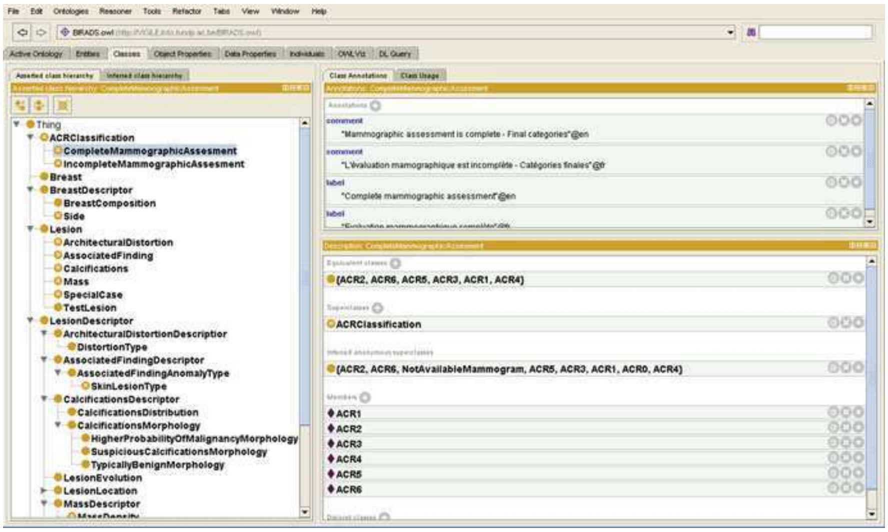
The annotation of mammography for breast cancer diagnosis is a good case to explain how the integration of both frameworks will be valuable. At the same time, it is evident that the case can be easily transposed to other cases of medical image annotation tasks, changing the domain of application. For this particular case, spatial segments are used to delimit what was identified by specialists and domain concepts are used to annotate the segments, describing the medical diagnosis. For this purpose, an ontology was created to explicitly specify each concept of the breast clinical domain. When an ontology is designed, it describes only one knowledge domain in order to be consistent and to provide its coherence, reuse, compatibility with other ontologies and lessen the risk of duplicity and ambiguity (Tudorache et al. 2008).

The ontology is developed by means of ontologies modeling Protégé platform.[17] The considered ontology gives all information about the patient (name, state of health), the type, place and date of study performed for this particular patient, and all possible outcomes of the study. The specialized medical part of the ontology is based on the classification by The American College of Radiology (ACR) that established the Breast Imaging Reporting and Database System (BI-RADS) to guide the breast cancer diagnostic routine (D'Orsi et al. 2003). BI-RADS is a quality assurance tool designed to standardize mammo-graphic reporting, guide radiologists and refer physicians in the breast cancer decision making process, reduce confusion in breast imaging interpretations, facilitate outcome monitoring and patients management.

Calcifications in the system followed by the considered ontology are described according to size, morphology and distribution. The findings are then interpreted and an assessment rendered that includes the degree of suspicion for malignancy, and any pertinent recommendations. This ontology provides the BI-RADS categories that are used to standardize interpretation of mammograms among radiologists. The implementation of BI-RADS categories in the ontology is shown in Fig. 20.4.

The ontology also defines the overall general composition of the breast defining possible type, shape, location of lesions, types of tissue density that can be present in patients. When instantiated with particular data for a particular patient, domain concepts allow efficient interpretation of the data obtained during the study and facilitate decision making concerning the perspectives of treatment and follow-up treatment plan.

[17]Protégé ontologies modeling platform: http://protege.stanford.edu.


File Edit Ontologies Reasoner Tools Refactor Tabs View Window Help
BIRADS.owl
Active Ontology Entities Classes Object Properties Data Properties Individuals OWLViz DL Query
Asserted class hierarchy Inferred class hierarchy
Class Annotations Class Usage

```
Thing
  ACRClassification
    CompleteMammographicAssesment
    IncompleteMammographicAssesment
  Breast
  BreastDescriptor
    BreastComposition
    Side
  Lesion
    ArchitecturalDistortion
    AssociatedFinding
    Calcifications
    Mass
    SpecialCase
    TestLesion
  LesionDescriptor
    ArchitecturalDistortionDescriptor
      DistortionType
    AssociatedFindingDescriptor
      AssociatedFindingAnomalyType
        SkinLesionType
    CalcificationsDescriptor
      CalcificationsDistribution
      CalcificationsMorphology
        HigherProbabilityOfMalignancyMorphology
        SuspiciousCalcificationsMorphology
        TypicallyBenignMorphology
    LesionEvolution
    LesionLocation
    MassDescriptor
```

comment
"Mammographic assessment is complete - Final categories"@en
comment
"L'évaluation mamographique est incomplète - Catégories finales"@fr
label
"Complete mammographic assessment"@en

{ACR2, ACR6, ACR5, ACR3, ACR1, ACR4}

ACRClassification

{ACR2, ACR6, NotAvailableMammogram, ACR5, ACR3, ACR1, ACR0, ACR4}

ACR1
ACR2
ACR3
ACR4
ACR5
ACR6

Fig. 20.4 Implementation of BI-RADS standard categories in the ontology

Adapting an Application to Use the Framework

MedicalStudio supports segmentation and annotation, but they were stored in DICOM format, which was not supported by Yasmim because it was developed for very specific needs on the medical domain. In order to support DICOM, Yasmim's data model was carefully compared with DICOM's data model. Modifications have been incorporated in Yasmim's model, since DICOM is a standard and it cannot be easily modified. Therefore, the integration of MedicalStudio and Yasmim depended on how compliant Yasmim's model is to DICOM.

After the compatibility check, MedicalStudio stopped storing in a DICOM format and started storing in Yasmim. The images have been stored in the media repository, segments in the relational database, and annotations in the document database. An additional web service was developed to load all this data and generate a DICOM file on demand. The resulting file can then be distributed to other medical systems. The URI to generate a DICOM file is the following: http://[server-name/domain]/resources/image/[id];format=dicom.

The algorithm used by MedicalStudio to manage this format was migrated to the new web service and the platform started using this service. There were two advantages on this approach:

1. MedicalStudio would become less complex by migrating part of its source code to Yasmim, consequently reducing the maintenance cost.
2. Other medical applications would profit from the new web service, by simply reusing it to generate their DICOM files.

In order to access this and other services, the library LibcURL,[18] a client-side URL transfer library, was added to MedicalStudio allowing HTTP connections to the server.

The current MedicalStudio version implements only one kind of segmentation, the spatial one, and three kinds of annotations, which are:

1. *Property*: used to annotate low level features of the image.
2. *Description*: if necessary, some description of the segment can be added.
3. *Domain concept*: the most common annotation, since MedicalStudio implements an ontology, as described in "Breast Diagnosis Domain Representation".

The process of mammography screening is as follow:

1. *Visualization*: mammographies are analyzed, they are retrieved with the patient records from the PACS[19] and HIS[20];
2. *Lesion detection*: each visible lesion in the image is detected, spatially localized (manually or with automatic algorithm) and categorized in one of the 5 possible type of lesions;
3. *Lesion annotation*: for each lesion a set of standardized characteristics is entered, all theses characteristics are dependants on the type of the lesion, the whole set is organized in an ontology based on a medical standard called BIRADS;
4. *Automatic annotation*: alongside the manual user annotation, a set of automatic algorithms characterises the same lesion with low level descriptions;
5. *Reporting*: finally, a diagnosis report is generated from the whole set of annotations made on all images and sent to the hospital information system.

Analyzing that process, we can exposes the inputs and outputs and maps them to YASMIM services in the integrated architecture. Table 20.2 lists them all. For each input and output there is a related web service, in the second column, and the type of data that has been manipulated, in the third column.

Integrated with Yasmim, MedicalStudio has another immediate benefit, which is the possibility to support multiple domains. It would allow specialists from different medical specialties to analyze the same image, enabling multi-disciplinary diagnosis. It would also allow the annotation of other kinds of images, besides mammographies. And last but not least, all data and annotations will be accessible to other medical clients, that is a simple and standardized way in addition to the complex DICOM document format. That access will allow the development of very thin clients, such as web clients or even smart-phone and tablet clients.

[18]http://curl.haxx.se/libcurl/.

[19]PACS : Picture Archiving and Communication System, usually implemented with the DICOM standard. http://medical.nema.org.

[20]HIS : Hospital Information System, a "in-house" system, but HL7 standard starts spreading. http://www.hl7.org.

Table 20.2 Breast cancer diagnosis: inputs and outputs

Tasks		Service	Data type
1. Visualization			
Outputs:	Images	Get media	Media resource
	Patient records	Get annotations	Properties
2. Lesion detection			
Inputs:	Spatial segment	Save segment	Segment
	Type selection	Save annotation	Properties
		Save annotation	Domain concept
Outputs:	List of types	Get annotations	Domain concept
3. Lesion characterization			
Inputs:	Characterization	Save annotation	Domain concept
Outputs:	List of characteristics	Get annotations	Domain concept
4. Automatic annotation			
Inputs:	User confidence	Save annotation	Domain concept
	Low level characteristics	Save annotation	Domain concept
Outputs:	Image information	Get annotations	Properties
5. Reporting			
Input:	Structured report	Save media	Resource
		Save annotation	Properties
Outputs:	Lesion and characteristics	Get media	Resource
		Get annotations	Properties
		Get annotations	Domain concept

Conclusion

This chapter presented a case study of a multimedia archiving framework fully implemented on the REST architectural style and applied on a medical imaging application. The architectural style represented by REST plays an important role on this evolution, precisely mapping the notion of *resource* with *media artifacts*, and being scalable to address the growing demand for media. A high level and a low level description of the architecture, our decisions concerning the design of the URIs, and a complete catalog of available services were presented.

The main concern about the use of Yasmim, at the moment, is a slightly decrease of performance due to network latency. Because every data is centralized on the server, the network is always considered. This issue might be addressed by saving some temporary data on the clients and synchronize them when necessary. It may also allow the tool off-line work.

Anyway, the adoption of Yasmim by MedicalStudio was positive because it contributed to reduce the complexity of MedicalStudio by migrating part of its features to Yasmim and, consequently, transforming it in a distributed application. Several specialists may have access to the repository at any time and place, and contribute with new images, segments and annotations.

Yasmim is an open source project, under Apache License 2.0.[21] It is maintained in the context of the 3D Media Project.[22]

References

Anderson, C., Lehnardt, J., Slater, N.: CouchDB: The Definitive Guide. O'Reilly Media Inc., Sebastopol, CA, USA (2010)

D'Orsi C.J., Bassett L.W., Berg W.A.: Breast Imaging Reporting and Data System: ACR BI-RADS-Mammography (ed 4), Reston, VA, American College of Radiology (2003)

Fielding, R.: Architectural styles and the design of network-based software architectures. Ph.D. thesis, University of California, Irvine (2000)

Gruber, T.R.: A translation approach to portable ontology specifications. Knowledge Acquisition 5, 199–220 (1993)

Kompatsiaris, Y., Hobson, P.: Introduction to semantic multimedia. In: Semantic Web Services: Concepts, Technologies, and Applications, chap. 1, pp. 3–13 (2008)

Richardson, L., Ruby, S.: RESTful Web Services. O'Reilly Media Inc., Sebastopol, CA, USA (2007)

Shapiro, L.G., Stockman, G.C.: Computer vision. Prentice Hall. (2001)

Trevisan, D., Nicolas, V., Macq, B., Nedel, L.: Medicalstudio: A medical component-based framework. In: Workshop de Informatica Medica - WIM (2007)

Tudorache, T., Noy, N.F., Tu, S.W., Musen, M.A.: Supporting collaborative ontology development in Protégé. In: Seventh International Semantic Web Conference, Karlsruhe, Germany, Springer. (2008)

[21] http://www.apache.org/licenses/LICENSE-2.0.html.

[22] http://mediatic.multitel.be/platforms/3dmedia.html.

Chapter 21
Metadata Architecture in RESTful Design

Antonio Garrote Hernández and María N. Moreno García

Abstract Metadata is a key component of the REST architecture that can be used to provide additional information about web resources. The ultimate goal of metadata is to transform web resources into self describing information units that can be automatically processed by software agents. We review the main options present in the HTTP standard to provide metadata for web resources. We also review the main mechanisms proposed by standard organizations like the W3C and the IETF as well as by groups of practitioners to provide additional ways of associating metadata to resources. The connection between metadata and semantic web technologies is also explored. Finally the notion of resource and metadata discovery is also introduced and the main discovery technologies are reviewed.

Introduction

Metadata is one of the data elements in the REST web architectural style along with resources, resource identifiers, representations and control data (Fielding and Taylor 2000). In this context, metadata can be defined as "machine understandable information about web resources" (Berners-Lee 1998). This brief definition remarks the importance of metadata as the element of RESTful design enabling automatic processing of web resources. This aspect is often overlooked in the design of RESTful web services where the role of metadata is many times restricted to provide information about the syntax used in the resource representation. This "representation metadata" encoded as a media data type in a HTTP header is exchanged between HTTP parties in the content negotiation process in order to select a suitable representation for a certain web resource.

A. Garrote Hernández (✉)
University of Salamanca, Avenida Italia 29 4-A, Plaza de los Caídos,
s/n, 37008, Salamanca, Spain
e-mail: agarrote@usal.es; antoniogarrote@gmail.com

Nevertheless, effective metadata for a resource should not be restricted to the automatic selection of the parsing mechanism for a resource representation. It should provide a full description of the semantics of the resource that would make possible for a HTTP agent to automatically choose a way of processing the resource to accomplish some kind of functionality, sometimes different from the original functionality devised by the creator of the resource. The ultimate goal of the metadata layer is to transform web resources into self describing information units (Berners-Lee 1998). This same aim can be found in the core of metadata proposals like the W3C's semantic web stack of technologies or the microformats initiative.

From a RESTful point of view, the main concern about the resource metadata layer is to find suitable ways of integrating this new kind of data into the architectural components of the REST style of building web systems. Metadata must be regarded as any other kind of resource data, therefore it must be exposed in the expected manner to REST components and connectors.

On top of this RESTful foundation, further metadata based features can be built, for instance, in the same way HTTP provide a common mechanism to access resources, metadata authors can agree in the vocabulary used in the metadata of a resource and in the ways these metadata are encoded into resources. These features have the potential to grant major advantages in web design, like improved data interoperability and better functional and conceptual reusability of web resources.

Metadata in RESTful web services is getting increasingly important as data APIs built following RESTful architectural principles are becoming a central component in many modern web applications. These APIs are facing the same data interoperability and reusability challenges that metadata have the potential to solve.

Metadata in the Hyper Text Transfer Protocol

The HTTP protocol makes metadata a first class object in the protocol specification. The main place to store metadata in HTTP messages is the collection of HTTP headers sent in every HTTP request and response.

Entity headers can be classified in different categories. Some headers contain meta information about the representation of the requested resource being transferred in the entity body of the HTTP response. The most important representation HTTP header is the "Content-Type" response header that specifies the media type for the representation. A different kind of entity headers expose meta data about the resource rather than the representation being transferred. For example the "Allow" HTTP header contains the list of HTTP methods supported by the resource. Finally, control data headers contain information necessary for the correct interaction between client and server. This information is not directly related to the representation of the resource. The Cache-Control header is an example of this kind of headers.

The main mean to provide the semantics of the resource representation retrieved by the client is the media type returned in the "Content-Type" HTTP header. This header is used in the context of the content negotiation mechanism specified in the HTTP protocol (Fielding et al. 1999). Using this mechanism a HTTP agent can expose the list of preferred representations it is willing to accept using the "Accept" HTTP header and the HTTP server can return the list of available representations for the resource using the "Content-Type" HTTP header. Examining this information both parties can agree which representation for the resource will be finally transferred from server to client.

Each media type imposes a certain syntax for the representation of the resource and hints some of the semantics of the resource being requested. media types can be classified into application specific media types and generic media types (Allamaraju 2010). Different media types provide a different degree of semantic information about the resource. Application specific media types specify well defined semantics for the representation of the resource encoded in the HTTP entity body. For example, the media type "image/jpg" provides enough semantics for HTTP agents to process the HTTP resource representation and visualize the entity data according to the specification of the JPEG image format. Application specific media types often have a very limited support for adding arbitrary metadata about the resource being encoded in the provided representation. On the other hand, media type headers for more generic media types, like "application/xml" or "application/json" provide very little semantic information about the resource but the format of the associated representation can contain any kind of metadata and information about the resource being retrieved.

A common practice for better describing the semantics of a representation is to provide an augmented media subtype in the Content-Type HTTP header. This way a particular vocabulary and semantics are stated to be used besides a generic encoding mechanism. media types like "image/svg+xml" or "application/atom+xml" give the agent a better understanding of the semantics of the representation built on top of a generic description mechanism (Allamaraju 2010).

The list of public media types is supported by IANA and is publicly available `http://www.iana.org/assignments/media-types/`. When designing a RESTful API is a good practice to look for a public representation format that suits the resources being exposed through the API. If no public media type matches the intended use of the representation at the application level, designers can consider the creation of new media types that will give HTTP agents a hint of the semantics for the provided representation of the resource.

The support for media types in the HTTP protocol metadata provides a mechanism to transfer the syntax and, to a certain degree, the semantics of the resource representation together with the representation in HTTP messages. Taking this into account, HTTP messages can be considered to be self-describing. Any HTTP connector can inspect the metadata of the HTTP message and take decisions about how to process the content of the message as an opaque packet of data. In a similar way, HTTP agents can decide the best way to process the message data based on the semantics of the stated media type.

Nevertheless, the level of semantic description of the content enabled by the use of media types is not enough to automate complex tasks involving the processing of web resources. Media types just provide HTTP agents with a reference to the semantics of the representation but it does not support a mechanism for describing these semantics. HTTP agents must have support in advance for the media type of the resource representation since it is impossible for agents to acquire support for an unknown type just from the media type declaration present in the HTTP "Content" header.

An additional problem is the rigidity of the media type standard to provide custom semantics for a specific resource. Private custom media type headers can be used for particular applications but their semantics cannot be automatically retrieved and processed by third party HTTP agents.

Different solutions to provide the semantics of the resource representation have been proposed. IETF RFC2068 (Fielding et al. 1997) of the HTTP protocol, superseded by IETF RFC2616, proposed the inclusion of a "Link" header (Conolly 1999) that could be used to link an associated resource to the resource representation being retrieved. This header has been used by different metadata retrieval proposals to associate metadata with a web resource. The use of an additional HTTP header has the advantage of not requiring the HTTP agent to retrieve the whole document in order to check and process the associated metadata. This can be accomplished with a single HEAD HTTP request that will retrieve the headers of the HTTP message. One major drawback of using the HTTP HEAD method is that it is not widely supported by server and client implementations of the HTTP protocol.

Using fixed, well known URIs where information about web resources in a domain could be retrieved, as proposed in IETF RFC5785 (Nottingham and Hammer-Lahav 2010), is another possible alternative for the association of metadata to resources that is being used in different metadata mechanisms.

WEBDAV extensions to the HTTP protocol introduced a different approach to the retrieval of metadata for a resource using an additional HTTP verb "PROPFIND" that make WEBDAV (Goland et al. 1999) enabled servers return all the metadata information associated with a resource.

Nevertheless, none of these mechanisms offers a solution for all the possible use cases involving the retrieval of metadata. In these cases, the common approach is to embed metadata in the resource representation or link the metadata from the representation if the resource representation supports hyperlinking. This approach can be problematic because HTTP agents must retrieve the full representation and process it to retrieve the metadata.

Metadata as a Formal System

There are different proposals to expose the description of the semantics of a web resource. Some of them have been proposed by standards organizations like the W3C, others have originated in the industry and among practitioners. As a consequence, designers of RESTful APIs face many different, often overlapping, options when choosing a mechanism to add semantic metadata to the representation of the resources exposed in web services.Nevertheless, there are some important points that must be taken into consideration and that should be addressed by any description mechanism.

First, semantic metadata should be regarded as a set of formal assertions about the resource being described (Berners-Lee 1998). Metadata must conform to a formal logic system with its own semantics that impose a certain trade-off between expressivity and processing complexity. This is a mandatory requirement to build truly extensible mechanisms for semantic description.

Any metadata proposal lacking formal soundness is unsuitable for the development of automated HTTP agents involving complex tasks like logic inference. More simple tasks like integration of resource information from different sources will also benefit from the coherence imposed by a formal description system.

Another major feature of a good semantic description mechanism is its capacity to interact with web technologies and follow RESTful architectural principles. One well known REST principle is "hypermedia as the engine of the application state" (Fielding 2000). It is important for any description mechanism to use the capacity of hyperlinking to reference descriptions from resources using URIs. URIs also offer a good namespace for creating unique identifiers for metadata that can be shared between agents in a web scale. The capacity of linking these descriptions from different resource descriptions, allows agents to retrieve the description of the semantics of a resource from the representation of the resource in a standard and RESTful way.

Finally, another important feature of any metadata description mechanism that must be taken into account is its openness and extensibility. The web itself is a system with a great degree of openness and extensibility arising from their basic components like the use of URIs and hyperlinks.

W3C standards for the semantic web meet all these requirements. Other proposals like the Microformats `http://microformats.org` initiative offer a simpler metadata mechanism at the price of limiting the expressivity and the extensibility of the solution. Nevertheless, microformats have been applied successfully to different application domains and have obtained broad adoption.

Embedding Metadata in Web Resources Using Microformats

Microformats, as a mechanism for the description of resource semantics, is an extension of the idea of semantic markup. Semantic markup is a set of design guidelines enforcing the use of HTML building blocks to express the meaning of the data contained in the document rather than the presentation information of those data.

To accomplish that goal, semantic markup principles enforce the use of HTML tags with precise semantics for each information element in the document. If there is no standard HTML tag with the required semantics for the information included in the document, a generic HTML container block like "div" or "span" can be used and the semantics of the information could then be added as the value of standard HTML attributes.

The Microformats proposal ultimate goal is to define standard ways of structuring HTML tag elements and property vocabularies in order to describe semantic information so it can be easily reused by humans in the design process of HTML documents and by software agents automatically processing web resources.

The main characteristics of the microformats initiative can be summarized as follows:

- Use of HTML structure plus a plain vocabulary to define semantics
- Community driven
- Embeddable in HTML, XHTML, Atom, RSS, and XML documents
- Focus on simplicity, reuse and minimalism

Currently, there is a list of ten microformats considered to be stable, including hCalendar for expressing calendar events, hCard used to represent people and organizations, or rel-license to state content licenses in a document.

The following example shows sample HTML code including hCalendar microformat markdown. The "event", "summary", "dtstart" and "location" property values are part of a controlled vocabulary used by the microformat to add semantic information to the data contained into standard span HTML tags. The structure of the HTML tags containing the vocabulary values in the class HTML properties is also prescribed by the hCalendar specification.

```
<span class="vevent">
 <span class="summary">The microformats.org site
 was launched</span>
 on <span class="dtstart">2005-06-20</span>
 at the Supernova Conference
 in <span class="location">San Francisco, CA, USA</span>.
</span>
```

In order to make microformatted HTML document self-describing, an HTML profile can be linked to the document describing the microformat, using the XMDP microformat itself. Profiles can be declared in the head of the HTML element using the "profile" attribute or the link tag with a "rel" attribute with value "profile". They can be just referenced in the HTML document body using a HTML anchor element with a "rel" property with value "profile".

Microformats provide simple mechanism to add semantics to HTML documents. They are easy to use with present technologies and have gained wide adoption. Unfortunately, microformats have important limitations. The main issue of the Microformats proposal is the use of plain literals to express the properties and relations of the HTML data. A literal used in one microformat to express a property can collide with any other use of the same literal in a different microformat with a similar or completely different meaning. Literal properties are not unique and they must be defined in a single flat namespace. The use of URIs would have avoided this problem since XML namespaces could be used to prevent name collisions among metadata identifiers but present important difficulties for their integration into plain HTML documents.

Another problem with the microformats is extensibility. It is impossible to add a custom microformat to a HTML document. The only description mechanism available for metadata is XMDPP and at the present moment is more suitable for documenting microformats for humans than for description of arbitrary microformats that could be automatically parsed by agents. Besides, the lack of unique identifiers for properties makes difficult to reuse properties between different microformats.

Resource Description Framework in Attributes and the W3C Semantic Stack of Technologies

Resource Description Framework in attributes (RDFa) (Pemberton et al. 2008) is the standard mechanism proposed by the W3C to embed semantic metadata into XHTML and HTML (Adida et al. 2010) documents. It has a similar aim to the Microformats proposal but is built on top of the stack of semantic technologies proposed by the W3C.

The foundation of the W3C semantic specifications is the Resource Description Framework (RDF) (Beckett 2004). RDF provides a mechanism to make statements about resources, understanding as a resource everything that can be identified using a URI (Miller and Manola 2004). RESTful resources are just a small subset of the resources that can be addressed in RDF. RDF is simultaneously many things:

- An abstract data model to describe metadata as a labeled graph of resources and properties
- An extensible vocabulary for data description based on the use of URIs
- A formal system with well defined semantics

RDF introduces the notion of properties, identified by URIs, that serve as a relation between resources acting as subject and object of the property. A collection of these statements, known as triples, defines a graph of relations between resources that can be serialized to different concrete syntaxes: XML, N3, TTL, etc.

RDFa is a standard proposal for a concrete syntax for the RDF data model that makes possible embedding RDF graphs in XHTML and other XML based documents. The final result is similar to the microformats proposal with some important differences:

- RDFa properties are stated as full URIs instead of plain literals. RDFa map URIs to literal strings using XML namespaces to build "compact URIs" or CURIES. The use of XML namespaces in the name of the relations prevents the clash between relation names. It also makes possible to reuse existing RDF vocabularies and allows an easy extension of the assertions contained in RDFa annotated XML documents.

- RDFa is currently defined on top of XHTML since it requires the extensibility capacities of this language. RDFa annotations can be used in HTML documents but these documents will not validate. Specification of RDFa annotations for HTML documents is an undergoing effort.

Since RDFa is just an encoding for RDF, the whole stack of semantic technologies standardized by the W3C can be used to provide semantics for the resource representation.

Schema and ontology languages like RDFS and OWL can be used to describe and link the description of the metadata used in the RDFa annotated document. For instance, vocabularies like the Dublin Core Metadata Initiative vocabulary `http://dublincore.org/`, or Friend of a Friend (FOAF) `http://www.foaf-project.org/` can be used in RDFa annotated documents in a similar way as microformats, like XFN or rel-license, are used in annotated HTML documents.

The following listing shows the example previously introduced for the hCalendar microformat modified to use RDFa and the iCalendar vocabulary.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML+RDFa 1.0//EN"
         "http://www.w3.org/MarkUp/DTD/xhtml-rdfa-1.dtd">
<html xmlns:cal="http://www.w3.org/2002/12/cal/ical#">

<span typeof="cal:Vevent">
 <span property="cal:summary">The microformats.org site
   was launched</span>
 on <span property="cal:dtstart" content="20050620">
   2005-06-20</span>
 at the Supernova Conference in
 <span property="cal:location">San Francisco, CA, USA</span>.
</span>
```

The vocabulary referenced by the URI “http://www.w3.org/2002/12/cal/ical#” has been added as a XML namespace declaration in the HTML tag. RDF properties from this namespace have been used as values for HTML attributes like “instanceof” or “property”. The values are specified as CURIES using the prefix “cal” previously declared in the XML namespace declaration.

RDFa tries to maintain the appeal of the Microformats proposal as a simple mechanism for adding semantics to XHTML documents while preserving the formal semantics and data model of the W3C stack of semantic technologies.

Extracting Metadata from Representations Using Transformations

Microformats and RDFa propose an approach to the description of metadata consisting of embedding metadata in the resource representation. Once a HTTP agent has retrieved the annotated representation, it can use a well defined algorithm to extract the actual metadata from the representation. These metadata can link to additional metadata, for example, a document containing the OWL description of the class and properties used to annotate the representation.

Gleaning Resource Description from Dialect of Languages (GRDDL) (Connolly 2007) is a W3C recommendation describing an alternative mechanism to add metadata to web resources. The starting point of GRDDL is the existence of a variety of possible representations for web resources. Many of them are XML based: plain XHTML documents, Atom feeds, etc. In many occasions, modifying these representations to embed semantic metadata using microformats or RDFa is not possible. One possibility to add semantic metadata to the resource is to provide an additional representation for the resource containing only the metadata for the resource being exposed, for example, a RDF document, that can be retrieved by HTTP agents using HTTP content negotiation. This approach has the drawback of creating and maintaining the additional representation.

Resource authors using GRDDL link an algorithmic transformation capable of generating a faithful rendition of the XML representation in RDF, instead of directly linking the metadata of the resource. GRDDL recommended way of describing transformations is using XSL Transformations (XSLT).

Linking GRDDL transformations from XML based resource representations can be accomplished just adding the GRDDL namespace declaration to the document and adding a "grddl:transformation" property pointing at the transformation. Transformations for whole XML dialects can be linked using the "grddl:namespaceTransformation" property. XHTML documents can be used with GRDDL adding the GRDDL namespace as a metadata profile and linking the transformation using a link tag with a "transformation" value for the "rel" attribute.

One of the main use cases for GRDDL is to transform XHTML documents annotated using microformats into RDF representations using some equivalent vocabulary. The following example shows a variant of the hCal microformat example considered before. In this example, a GRDDL transformation renders the same RDF graph that was embedded in the RDFa annotated version.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
  "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
  lang="en">
  <head profile="http://www.w3.org/2003/g/data-view">
    <link rel="transformation"
          href="http://www.w3.org/2002/12/cal/glean-hcal"/>
  </head>
  <body>
    <span class="vevent">
     <span class="summary">The microformats.org site
     was launched</span>
     on <span class="dtstart">2005-06-20</span>
     at the Supernova Conference
     in <span class="location">San Francisco, CA, USA</span>.
    </span>
  </body>
</html>
```

GRDDL offers also a good opportunity for reusability. The GRDDL community has collected transformations for many existing microformats as well as for other non HTML based dialects e.g. the Atom format that can be directly linked by authors of these representations. GRDDL can be used by Microformats publishers as an easy way to provide an alternative representation for the metadata of a resource that can be used by agents working with W3C semantic technologies.

Resource and Metadata Discovery

Services discovery can be described as the process allowing two automated agents to start some kind of useful interaction. In the process both parties discover which kind of services are offered by the other. In the context of web resources, we can talk about two kind of discovery process: service discovery and descriptor discovery (Hammer-Lahav 2010). Service discovery deals with agents looking for services with a certain capability. Descriptor discovery involves a software agent trying to discover the capabilities supported by a resource. The availability of metadata is one of the services that can be detected in the service discovery mechanism, metadata themselves can also be used to make possible the discovery of other kind of capabilities as well as enabling service discovery.

In previous sections, different ways of adding metadata to a resource have been examined. Previously reviewed mechanisms like the embedding of metadata into resource representations using microformats and RDFa or linking a GRDDL transformation capable of generating metadata from a representation, present the problem of not being automatically discoverable by HTTP agents. Agents must obtain the full representation of the HTTP resource and process it in order to detect the presence of metadata description mechanisms.

Metadata discovery protocols and specifications try to solve these limitations providing two features:

- Standard protocols for the automatic retrieval of resource’s metadata.
- Shared vocabularies for the description of resource services.

Discovery mechanisms must have certain desirable features that are also common to any other metadata mechanisms (Umbrich et al. 2009):

- Self declarative: the resource must be capable of linking the resource description.
- Direct accessible: the resource description must be retrievable without requesting the resource being described.
- Compliant with web architecture.
- Scale to web size.
- Extensible: the description mechanism must allow authors of resource descriptors to add arbitrary metadata in the description.
- Granular: a resource descriptor can be used to describe a single resource or a set of resources.

The Protocol for Web Description Resources (POWDER) is a W3C standard recommendation for the discovery of metadata associated with a web resource. POWDER documents consist of two different parts: an attribution block describing the author, date and validity of the description and a collection of "description resources" containing the actual metadata. A single POWDER document can contain metadata for different resources in a single domain. Each description resource is composed of two parts, a set of URIs being described and a collection of assertions. The assertions contain a collection of plain tags or a RDF fragment.

POWDER documents can be linked using the "describedby" property from the POWDER namespace. HTML documents can link a POWDER description using link tags located in the head of the document. In order to avoid the necessity of processing the whole resource representation, a different recommended mechanism to link a POWDER profile is to use the not official "Link" HTTP header.

An alternative mechanism for metadata discovery is the combination of the LRDD/XRD standards. Link-based Resource Descriptor Discovery (LRDD) protocol is an IETF draft standard proposal for linking easily discoverable metadata to web resources.

LRDD defines three different metadata sources:

- Hyperlinks using the "link" tag in the representation of a resource.
- The "Link" HTTP header.
- Host metadata situated in standard locations.

LRDD shares with POWDER the hyperlink and "Link" header mechanisms for metadata discovery. It also adds the possibility of inserting metadata at standard locations for each domain. The IETF "host-meta" standard proposal specifies a single point for each domain to add metadata for resources located at that domain. The entry point URI is built from the "host-meta" suffix added to the standard "/.well-known/" prefix defined in the IETF RFC 5785. LRDD profiles collected from these three link sources must be described using the OASIS standard draft Extensible Resource Descriptor (XRD) as the format for the metadata of the described resources.

Conclusions

The current state of the metadata architecture in the design of RESTful web services is still a work in progress.

As RESTful APIs are becoming more and more usual and a higher degree of automation and interconnection is required, the necessity of a standard metadata layer is becoming more evident. In this chapter we have reviewed some of the main technologies trying to address the architectural issues introduced by the integration of semantic metadata in the HTTP protocol.

Four main techniques have been introduced to associate metadata with resource representations:

- Providing metadata as an alternative representation for the resource that can be retrieved using content negotiation.
- Embedding metadata within the resource representation.
- Linking metadata annotations from the resource representation.
- Linking metadata annotations from the headers of the HTTP message or a well known URI location.

Microformats have been used as a simple mechanism to embed metadata in HTML documents. Microformats is the most extended technology to add explicit semantics to HTML representations but this technology is lacking in extensibility and presents a serious drawback due to the use of a flat namespace to describe properties instead of standard XML namespaces and URIs. To solve these problems, RDFa presents an alternative mechanism to embed metadata in XHTML documents in a compliant way with W3C standards for the semantic web. RDFa allows publishers of web resources to add the full potential of semantic web technologies to their service APIs, like the ontology description language OWL and standard vocabularies like FOAF at the same time that it preserves the simplicity and low entry barrier of the Microformats proposal.

A bridge between both semantic annotation mechanisms can be found in the GRDDL W3C recommendation. GRDDL provides the means for linking an algorithmic transformation to a resource representation that will render as a result of its application, the equivalent RDF triples graph. GRDDL transformations can be reused by resource publishers and a whole collection of GRDDL transformations for many microformats is already available. GRDDL makes possible the integration of annotation mechanisms, metadata vocabularies and description mechanisms.

Metadata discovery is another open problem in order for autonomous HTTP agents to be able to identify the available metadata in the services exposed by API providers. Automatic discovery of these metadata will open new ways of interaction between agent and servers. The POWDER W3C recommendation and the XRD/LLDR protocol stack try to offer solutions to this problem, specifying linking mechanisms that do not require the HTTP client to retrieve and process the full representation of the resource. Metadata can be linked to the HTTP message using the Link HTTP header that can be retrieved using the HEAD HTTP method without downloading the HTTP message body or can be placed into well defined standard URIs that can be queried by clients, for example, in the ".well-known/host-meta" path. They also define standard ways of adding arbitrary metadata to services and the mechanism for its retrieval.

Metadata is already making possible the automatic interaction between HTTP agents in web protocols like the OAuth authentication mechanism. In the nearly future, better metadata support in RESTful APIs will make possible to automate new kind of interactions offering important benefits to users. The emergent properties as well as the interoperability capacities offered by semantic metadata will also make possible to build more robust HTTP agents and use these APIs in new ways not anticipated by their original designers.

References

B. Adida, M. Birbeck, and S. Pemberton. HTML+RDFa 1.1, support for rdfa in html4 and html5. W3C working draft, W3C, October 2010. http://www.w3.org/TR/rdfa-in-html/.

S. Allamaraju. *RESTful Web Services Cookbook*. O'Reilly, February 2010.

D. Beckett. RDF/xml syntax specification (revised). W3C recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/.

T. Berners-Lee. Design issues of web architecture. 1998.

D. Connolly. Gleaning resource descriptions from dialects of languages (GRDDL). W3C recommendation, W3C, September 2007. http://www.w3.org/TR/2007/REC-grddl-20070911/.

H. Conolly. An Entity Header for Linked Resources, October 1999.

R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, Irvine, California, 2000.

R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2068 (Proposed Standard), January 1997. Obsoleted by RFC 2616.

R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785.

R. T. Fielding and R. N. Taylor. Principled design of the modern web architecture. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 407–416, New York, NY, USA, 2000. ACM.

Y. Goland, E. Whitehead, A. Faizi, S. Carter, and D. Jensen. HTTP Extensions for Distributed Authoring – WEBDAV. RFC 2518 (Proposed Standard), February 1999. Obsoleted by RFC 4918.

E. Hammer-Lahav. LRDD: Link-based Resource Descriptor Discovery, Draft rev 6. Internet Draft, May 2010.

E. Miller and F. Manola. RDF primer. W3C recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-rdf-primer-20040210/.

M. Nottingham and E. Hammer-Lahav. Defining Well-Known Uniform Resource Identifiers (URIs). RFC 5785 (Proposed Standard), April 2010.

S. Pemberton, B. Adida, S. McCarron, and M. Birbeck. RDFa in XHTML: Syntax and processing. W3C recommendation, W3C, October 2008. http://www.w3.org/TR/2008/REC-rdfa-syntax-20081014.

J. Umbrich, M. Hausenblas, E. Hammer-Lahav, and E. Wilde. Discovering resources on the web. DERI technical report, DERI, August 2009.

Chapter 22
RESTful Services with Lightweight Machine-readable Descriptions and Semantic Annotations

Jacek Kopecký, Tomas Vitvar, Carlos Pedrinaci, and Maria Maleshkova

A little semantics goes a long way.

– Jim Hendler

Abstract REST was originally developed as the architectural foundation for the human-oriented Web, but it has turned out to be a useful architectural style for machine-to-machine distributed systems as well. The most prominent wave of machine-oriented RESTful systems are Web APIs (also known as RESTful services), provided by Web sites such as Facebook, Flickr, and Amazon to facilitate access to the services from programmatic clients, including other Web sites.

Currently, Web APIs do not commonly provide machine-processable service descriptions which would help tool support and even some degree of automation on the client side. This chapter presents current research on lightweight service description for Web APIs, building on the HTML documentation that accompanies the APIs. descriptions. HTML documentation can be annotated with a microformat that captures a minimal machine-oriented service model, or with RDFa using the RDF representation of the same service model. Machine-oriented descriptions (now embedded in the HTML documentation of Web APIs) can also capture the semantics of Web APIs and thus support further automation for clients. The chapter includes a discussion of various types and degrees of tool support and automation possible using the lightweight service descriptions.

Introduction

This book deals extensively with RESTful services and Web APIs,[1] a machine-oriented part of the Web. In contrast to other technologies focused on services or distributed computing, RESTful services seldom come with machine-processable

[1]In this chapter, we use the terms such as “Web API”, “RESTful service” etc. interchangeably.

J. Kopecký (✉)
Knowledge Media Institute, Open University, Walton Hall, Milton Keynes, MK7 6AA, UK
e-mail: j.kopecky@open.ac.uk

service descriptions that would enable client-side tool support and even some degree of automation.

The reasons for the reluctance of Web API providers to create and maintain machine-processable service descriptions likely stem from the *DRY* principle (*Don't Repeat Yourself*): on one level, service providers already produce HTML documentation for their services and they do not want to maintain another description; and on another level, RESTful systems (are supposed to) use "hypertext as the engine of the application state" – the clients should be guided by the hypertext structure of the resources of a given service, rather than by some external service descriptions. Moreover, there are currently no widely-accepted standards for machine-processable descriptions of RESTful services, increasing the uncertainty about adopting heavyweight technologies such as WADL because the effort might be wasted if another technology becomes the standard.

In this chapter, we show a lightweight approach to describing RESTful services in a machine-processable form. The approach builds on a minimal service model that covers the important aspects of the structure of Web services. We show two simple ways to structure the existing HTML service documentation – a microformat called hRESTS and a generic standard form called RDFa – to provide machine-processable service descriptions with no duplication of content.

On top of machine-readable service descriptions, we demonstrate a straightforward application of Semantic Web Services approaches for further advanced machine processing and automation. In particular, we capture service semantics using the W3C standard SAWSDL (Semantic Annotations for WSDL and XML Schema 2007) and the W3C-acknowledged research proposal WSMO-Lite (Fensel et al. 2010), in the spirit of earlier works called WSDL-S (Akkiraju et al. 2005) and SA-REST (Sheth et al. 2007).

The aim of employing semantic technologies is to help with the following tasks: *discovery* matches known Web services against a user goal and returns the services that can satisfy that goal; *ranking* orders the discovered services based on user requirements and preferences so the best service can be selected; *composition* puts together multiple services when no single service can fulfill the whole goal; *invocation* then communicates with a particular service to execute its functionality; and *mediation* resolves any arising heterogeneities.

Our research, whose results are presented in this chapter, is driven by the following conclusions drawn from previous works on service description and from the progress towards the Web of Data:

- Semantics are essential to reach a minimum level of automation during the life-cycle of services;
- Any solution to publishing services that aspires to be widely adopted should build upon the various approaches and standards used on the Web, e.g. RDF, SPARQL and Web APIs;
- Linked Data principles are important for publishing large amounts of semantic data, both for human and machine consumption;

- On the Web, lightweight ontologies together with the possibility to provide custom extensions prevail against more complex models;
- The annotation of service descriptions should be simplified as much as possible.

This chapter has the following structure: in "Modeling RESTful Services and Web APIs" (page 475), we discuss the structure of RESTful services and Web APIs, and we substantiate an operation-oriented view of services that leads to the formal service model defined in "Minimal Service Model" (page 480). "hRESTS: Microformat for Service Descriptions" (page 483) and "Service Description with the Minimal Service Model and RDFa" (page 488) specify two approaches – a microformat and RDFa – for annotating existing service documentation so that it becomes machine-processable according to the minimal service model. In "Service Semantics with WSMO-Lite" (page 492), we extend the basic service descriptions with semantic information and we discuss what automation can be achieved with such enhanced descriptions. Finally, "Tools and Implementations" (page 498) deals with implementations and tools that can be built to create and process lightweight service descriptions; the section details a service registry iServe and an editor and annotator tool SWEET. "Summary" (page 504) summarizes the chapter.

Modeling RESTful Services and Web APIs

In their structure and behavior, RESTful services can be very much like common Web sites (Richardson and Ruby 2007). From the Architecture of the Web (Architecture of the World Wide Web 2004) and from the REST architectural style, we can extract the following concepts inherent in RESTful services:

- A *resource*, identified by a URI which also serves as the endpoint address where clients can send requests.
- Every resource has a number of *methods* (in HTTP, the most-used methods are GET, POST, PUT and DELETE) that are invoked by means of request/response message exchanges.
- The messages can carry *hyperlinks*, which point to other resources and which the client can navigate when using the service.
- A hyperlink can simply be a URI, or it can be a *form* which specifies not only the URI of the target resource, but also the method to be invoked and the structure of the input data.

Note that even though we talk about RESTful services, the architecture of the Web contains no formal concept of a *service* as such. On the Web, a service is a group of resources; such grouping is useful for developing, advertising and managing related resources.

While the resources of the service form a hypermedia graph, the interaction of a client with a RESTful service is a series of operations where the client sends a request to a resource and receives a response that may link to further useful

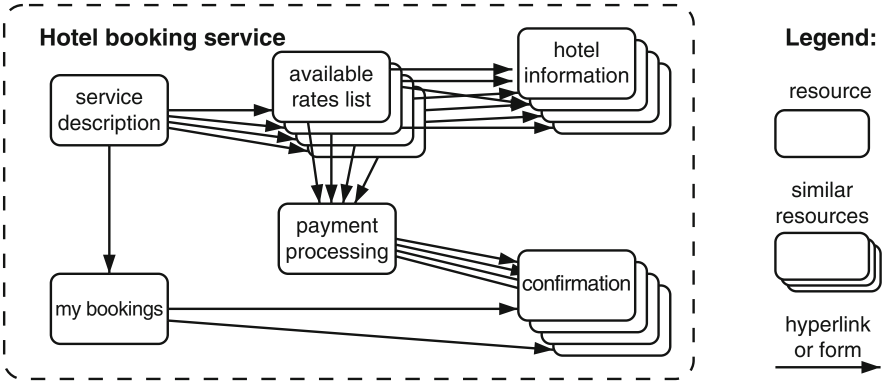


Fig. 22.1 Structure of an example hotel reservation service

resources. The hypermedia graph (the links between resources) guides the sequence of operation invocations, but the meaning of a resource is independent of where it is linked from; the same link or form, wherever it is placed, leads to the same action. Therefore, the operations of a RESTful service can be considered independently from the graph structure of the hypertext.

In this chapter, we build upon the independence of operations and hypertext. To illustrate this independence, and to show how a programmatic client interacts with a service effectively by invoking a set of operations, treating the hypertext links as data, the subsections below describe an idealized hypertext hotel reservation service with a RESTful API. Section "Example Hotel Booking Service, Viewed as Hypertext" (page 476) describes the service as a hypertext graph of resources, "Turning Hypertext into Operations" (page 478) turns to view the service as a set of operations, and then in "HTML Description of the Example Service" (page 479), we discuss how such a RESTful API would typically be documented in HTML. API documentation is the basis for our lightweight service descriptions, as detailed further in this chapter.

Example Hotel Booking Service, Viewed as Hypertext

Figure 22.1 illustrates an example RESTful hotel booking service, with its resources and the links among them. Together, all these resources form the hotel booking service; however, the involved Web technologies actually work on the level of resources, so *service* is a virtual term here and the figure shows it as a dashed box.

The "service description" (page 479) is a resource with a stable address and information about the other resources that make up the service. It serves as the initial entry point for client interaction. In a human-oriented Web application, this would be the homepage, such as `http://hotels.example.com/`.

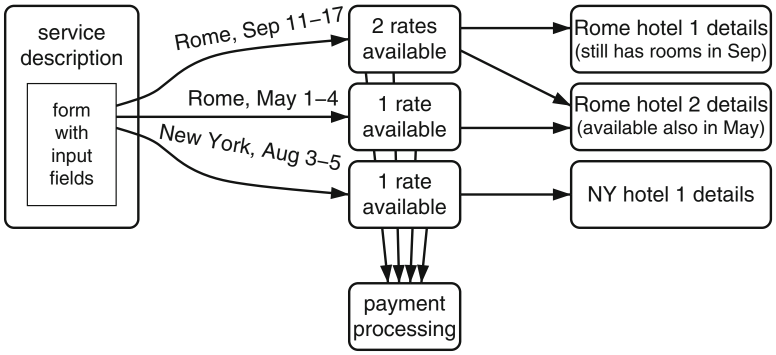


Fig. 22.2 Detail of example hotel reservation service resources for hotel search and hotel details

The existence of such a stable entry point lowers the coupling between the service and its clients, and it enables the evolution of the service, such as adding or removing functionality. A client need only rely on the existence of the fixed entry point, and it can discover all other functionality as it navigates the hypermedia. (However, in many cases, a programmatic client is programmed against a given service description before it uses the service, making it harder to react dynamically to changes of the service. This is especially true in service-description-driven technologies such as most of tools for WS–∗ Web services, but it is also common with access libraries for Web APIs, such as the many "API kits"[2] for the Flickr API.)

The service description resource of our example service contains a form for searching for available hotels, given the number of guests, the start and end dates and the location. The search form serves as a parametrized hyperlink to search results resources that list the available rates, as detailed in Fig. 22.2; one resource per every unique combination of the input data. The form prescribes how to create a URI that contains the input data; the URI then identifies a resource that returns the list of available hotels and rates for the particular inputs. As there is a large number of possible search queries, there is also a large number of results resources, and the client does not need to know that all these resources are likely handled by a single software component on the server.

The search results are modeled as separate resources (as opposed to, for instance, a single data-handling resource that takes the inputs in a request message of a POST method), because it simplifies the reuse of the hotel search functionality in other services or in mashups (lightweight compositions of Web applications), and it also supports caching of the results. Creating the URIs for individual search results resources and retrieving the results (with HTTP GET) is easier in most programming frameworks than POSTing the input data in a structured data format to a single Web resource that would then reply with the list of available hotels and rates.

[2]See http://www.flickr.com/services/api/#kits.

Search results are presented as a list of concrete rates available at the hotels in the given location, for the given dates and the number of guests, as also shown in Fig. 22.2. Each item of the list contains a link to further information about the hotel (e.g. the precise location, star rating, guest reviews and other descriptions), and a form for booking the rate, which may take as input the payment details (such as credit card information) and an identification of the guest(s) who will stay in the room. The booking data is submitted (POSTed) to a payment resource, which processes the booking and redirects the client to a confirmation resource, as shown in Fig. 22.1. The content of the confirmation can serve as a receipt.

The service description resource also contains a link to "my bookings", a resource that lists the bookings of the current user (this would require authentication). This resource links to the confirmations of the bookings done by the user. With such a resource available to them, client applications do not need to have a local store for the information about performed bookings.

The confirmation resources may further provide a way of canceling the reservation (not shown in the pictures, could be implemented with the HTTP DELETE method).

Turning Hypertext into Operations

So far, our description of the example hotel reservation service has focused on the hypermedia aspect: we described the resources and how they link to each other. Alternatively, we can also view the service as a set of operations available to the clients – as an API.

The resources of the service (the *nouns*) form a hypermedia graph (shown in Fig. 22.1). The interaction of a client with a RESTful service is a series of operations (the *verbs* or *actions*) where the client sends a request to a resource and receives a response that may link to further useful resources. Importantly, the links need not be only simple URIs, but they can also be *input forms* that indicate the URI, the HTTP method, and the input data.

The graph nature of a hypermedia service guides the sequence of operation invocations, but the meaning of a resource is independent of where it is linked from; the same link or form, wherever it is placed, always means the same operation. Therefore, the operations of a RESTful service can be considered independently from the graph structure of the hypertext.

In Fig. 22.3, we extract the operations present in our example service. The search form in the service description (homepage) represents a search operation, the hotel information pages linked from the search results can be viewed as an operation for retrieving hotel details, the reservation form for any particular available rate becomes a reservation operation, and so on.

An operation-oriented view on RESTful services brings them closer to common programming environments; it is a natural view for programmers of specialized client applications.

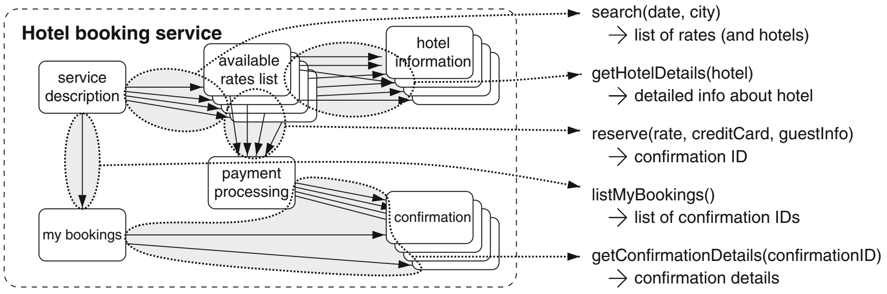


Fig. 22.3 Operations of the example service

HTML Description of the Example Service

Web APIs, or indeed services of any kind, need to be described in some way, so that potential clients can know how to interact with them. While Web applications are self-describing to their human users, Web services are designed for machine consumption, and someone has to tell the machine how to consume any particular service.

Public RESTful services are universally described in human-oriented documentation (for instance, see Flickr API[3] and Amazon Simple DB[4]) using the general-purpose Web hypertext language HTML. Typically, such documentation will list the available operations (calling them API calls, methods, commands etc.), their URIs and parameters, the expected output and error conditions and so on; it is, after all, intended as the documentation of a programmatic interface.

The following might be an excerpt of a typical operation description for our example hotel reservation service:

ACME Hotels service API
Operation `getHotelDetails`

Invoked using the method GET at `http://example.com/h/id`
Parameters: `id` - the identifier of the particular hotel
Output value: hotel details in an `ex:hotelInformation` document

In HTML, the description can be captured as shown in Listing 22.1.

Such documentation has all the details necessary for a human to be able to create a client program that can use the service. We can amend the textual documentation to tease out these technical details and make them accessible to machine processing and tools; the following sections of this chapter show two approaches that use a common minimal service model and annotate the HTML documentation to be machine-processable.

[3] http://flickr.com/services/api.

[4] http://docs.amazonwebservices.com/AmazonSimpleDB/latest/DeveloperGuide.

```
<h1>ACME Hotels service API</h1>
<h2>Operation <code>getHotelDetails</code></h2>

<p> Invoked using the method GET at <code>http://example.com/h/{id}</code> <br/>
  <strong>Parameters:</strong>
    <code>id</code> – the identifier of the particular hotel <br/>
  <strong>Output value:</strong> hotel details in an
    <code>ex:hotelInformation</code> document
</p>
```

Listing 22.1 Example HTML service description

In the hypertext of the example service, the service has five operations but only two are directly accessible from the service description resource. All five operations can be described in a single HTML document, but the client would not know any concrete hotel identifiers to invoke `getHotelDetails()` before it does its first hotel search; similarly, the client won't have any confirmation ID to invoke `getCofirmationDetails()` before it makes its first reservation. While using the service, the client may save hotel or confirmation identifiers and use them later to invoke these operations without going through availability searches or the list of "my bookings"; this behavior is roughly equivalent to how bookmarks work in a Web browser.

Minimal Service Model

The client-side independence of operations from the resource and hypermedia structure of a RESTful API allows us to view RESTful services through a minimal service model with terminology adopted from WSDL, as shown in Table 22.1.

The structure of the resulting service model is shown in Fig. 22.4. A Web service has a number of operations, each with potential input and output messages, and underlying the operations is a hypertext graph structure where the outputs of one operation may link to other operations. This model captures the requirements for what we need to represent in a machine-readable description. The model is very similar in its structure to WSDL, only instead of hypertext, WS–$*$ services commonly use the terms "process" or "choreography" for the sequencing of operations.

As operations in the minimal service model correspond to HTTP methods on the resources of RESTful services, each operation description can specify a resource address (a URI or a parametrized URI template,[5]) the HTTP method (usually GET, POST, PUT or DELETE), and the input and output data formats. In principle, the

[5]URI templates are defined for instance in WSDL 2.0 HTTP Binding (Web Services Description Language (WSDL) Version 2.0: Adjuncts 2007) in Sect. 6.8.1.1.

Table 22.1 Mapping RESTful services into a minimal service model, using WSDL terminology

RESTful services	Minimal service model
Service *(a group of resources)*	Service
Resource	– *(mapped below, in conjunction with methods)*
HTTP method on a resource	Operation *(specifying a method and a resource address)*
Method request/response	Operation input/output message
Resource representations	– *(treated as message data)*
Hyperlink	– *(treated as part of message data)*

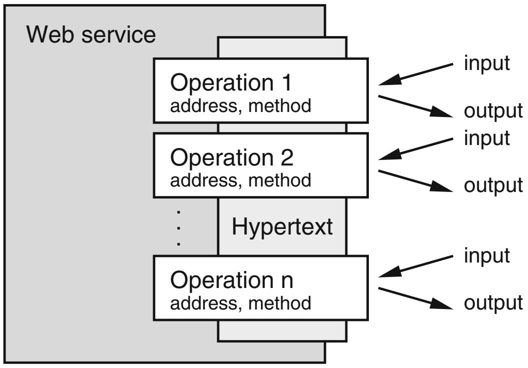


Fig. 22.4 Functional model of RESTful services, with the service, its operations and their input and output messages

output data format can be self-describing (self-description is a major property of Web architecture), but the API documentation should specify what the client can expect.

The input and output messages of the operations in the minimal service model correspond to the request and response of the HTTP methods on the resources of RESTful services. The messages can be described on a finer level of granularity, decomposed into message parts, which can be mandatory or optional. Modeling message parts is intended to support finer-grain discovery based on data structure instead of the message as a whole, mirroring the granularity of SAWSDL in XML Schema, and allowing to distinguish between mandatory and optional parts.

While at runtime the client interacts with concrete resources, the service description may present a single operation that acts on many resources (such as `getHotelDetails(hotel)` which operates on any hotel details resource), therefore an operation can specify an *address* as a URI template whose parameters are part of the operation's input data.

Listing 22.2 shows an RDFS realization of this service model, together with the operation properties described above. Concrete service descriptions (using syntaxes defined in "hRESTS: Microformat for Service Descriptions" (page 483)

```
@prefix hr:   <http://www.wsmo.org/ns/hrests#> .
@prefix rdf:   <http://www.w3.org/1999/02/22−rdf−syntax−ns#> .
@prefix rdfs:   <http://www.w3.org/2000/01/rdf−schema#> .
@prefix xsd:   <http://www.w3.org/2001/XMLSchema#> .

# classes and properties of the minimal service model
hr:Service  a  rdfs:Class .
hr:hasOperation  a  rdf:Property ;
  rdfs:domain  hr:Service ;
  rdfs:range  hr:Operation .
hr:Operation  a  rdfs:Class .
hr:hasInputMessage  a  rdf:Property ;
  rdfs:domain  hr:Operation ;
  rdfs:range  hr:Message .
hr:hasOutputMessage  a  rdf:Property ;
  rdfs:domain  hr:Operation ;
  rdfs:range  hr:Message .
hr:Message  a  rdfs:Class ;
  rdfs:subClassOf  hr:MessagePart .
hr:MessagePart  a  rdfs:Class .
hr:hasMessagePart  a  rdf:Property ;
  rdfs:domain  hr:MessagePart ;
  rdfs:range  hr:MessagePart .
hr:hasMandatoryPart  rdfs:subPropertyOf  hr:hasMessagePart .
hr:hasOptionalPart  rdfs:subPropertyOf  hr:hasMessagePart .

# operation properties for RESTful services
hr:hasAddress  a  rdf:Property ;
  rdfs:range  hr:URITemplate .
hr:hasMethod  a  rdf:Property ;
  rdfs:range  xsd:string .

# a datatype for URI templates
hr:URITemplate  a  rdfs:Datatype .

# RDFS properties commonly used with this service model
rdfs:isDefinedBy  a  rdf:Property .
rdfs:label  a  rdf:Property .
```

Listing 22.2 Minimal service model in RDFS (in Turtle syntax)

and “Service Description with the Minimal Service Model and RDFa” (page 488)) can be parsed into instances of this RDFS model and stored in a service registry (see “iServe: A Service Registry” (page 499)) for processing in various types of tools.

On top of the material properties defined in the model above, services, their operations, and messages can also have human-readable names, which can be attached in RDF using the `rdfs:label` property. Additionally, it is useful to include an `rdfs:isDefinedBy` link from a particular service described with this model back to the service’s HTML documentation; such a link will allow tools for example to show the relevant documentation snippets when a user browses the API (this would be similar to how JavaDoc snippets are shown in Java programming IDEs).

Interoperability with WS–∗ services Note that since WSDL descriptions can trivially be mapped into the same minimal model, it allows a single client framework to support WS–∗ and RESTful services without regard to their technological differences; this is especially true for semantic clients, discussed below in “Service Semantics with WSMO-Lite” (page 496).

Semantic annotations As shown in the following sections, the minimal service model is applied on HTML documentation of RESTful services to make it amenable to machine processing. The model identifies key pieces of information that are already present in the documentation, effectively creating an analogue of WSDL. As such, the model forms a basis for further extensions, where service descriptions are annotated with added information to facilitate further processing. One such extension is semantic annotations, meant to support powerful service discovery and even the application of AI technologies such as automated service composition.

Because the model is so similar to WSDL, we can adopt SAWSDL (Semantic Annotations for WSDL and XML Schema 2007) properties to add semantic annotations. SAWSDL specifies how to annotate service descriptions with semantic information. It defines the following three RDF properties:

- `modelReference` is used on any component in the service model to point to appropriate *semantic concepts* identified by URIs. SAWSDL speaks about semantic concepts in general, which is not to be confused with the specialized use of the term *concept* in some literature to denote what is called *class* in OWL; a model reference can point to any element of a semantic description.
- `liftingSchemaMapping` and `loweringSchemaMapping` are used to associate messages with appropriate transformations, also identified by URIs, between the underlying technical format such as XML and a semantic knowledge representation format such as RDF.

hRESTS: Microformat for Service Descriptions

In the preceding sections of this chapter, we have discussed the structure of RESTful services, viewed as sets of operations, and we have noted that RESTful services are universally described with HTML documentation, while providers seem reluctant to also create and maintain machine-oriented service descriptions for their RESTful services.

In this section, we introduce hRESTS, a microformat that can be used to structure the existing RESTful Web service documentation so that the key pieces of information are machine-processable. Microformats are an "adaptation of semantic XHTML that makes it easier to publish, index, and extract semi-structured information" (Khare and Çelik 2006), an approach for annotating mainly human-oriented Web pages so that selected information is machine-readable. On top of microformats, GRDDL (Gleaning Resource Descriptions from Dialects of Languages GRDDL 2007) is a mechanism for extracting RDF information from Web pages, particularly suitable for processing microformats. For instance, there are already microformats for contact information, calendar events, ratings etc.

Microformats take advantage of existing XHTML facilities such as the `class` and `rel` attributes to mark up fragments of interest in a Web page, making the fragments easily available for machine processing. For example, a calendar

```
1  <div class="service" id=”svc”>
2   <h1><span class="label">ACME Hotels</span> service API</h1>
3   <div class="operation" id=”op1”>
4    <h2>Operation <code class="label">getHotelDetails</code></h2>
5    <p> Invoked using the <span class="method">GET</span>
6     at <code class="address">http://example.com/h/{id}</code><br/>
7     <span class="input">
8      <strong>Parameters:</strong>
9      <span class="parameter mandatory">
10       <code class="label">id</code> – the identifier of the particular hotel
11      </span>
12     </span><br/>
13     <span class="output">
14      <strong>Output value:</strong> hotel details in an
15      <code>ex:hotelInformation</code> document
16     </span>
17    </p>
18 </div></div>
```

Listing 22.3 Example hRESTS service description

microformat marks up events with their start and end time and with the event title, and a calendaring application can then directly import data from what otherwise looks like a normal Web page. Further details on how microformats work can be found at microformats.org.

The hRESTS microformat is made up of a number of HTML classes that correspond directly to the various parts of the minimal service model. To help illustrate the detailed definitions of the hRESTS classes, in Listing 22.3 we show hRESTS annotations of the sample HTML service description shown in Listing 22.1.

In the following detailed definitions, we refer to RDF classes and properties from the service model (Listing 22.2) using the prefix `hr`.

The **`service`** class on block markup (e.g. `<body>`, `<div>`), as shown in the example listing on line 1, indicates that the element describes a service API. An HTML element with the class `service` corresponds to an instance of `hr:Serv-ice`. A service contains one or more operations and may have a label (see below).

The **`operation`** class, also used on block markup (e.g. `<div>`), indicates that the element contains a description of a single Web service operation, as shown in the listing on line 3. An element with this class corresponds to an instance of `hr:Operation`, attached to its parent service with `hr:hasOperation`. An operation description specifies the address and the method used by the operation, and it may also contain description of the input and output of the operation, and finally a label.

The **`address`** class is used on textual markup (e.g. `<code>`, shown on line 6) or on a hyperlink (`<a href>`) and specifies the URI of the operation, or the URI template in case any inputs are URI parameters. Its value is attached to the operation using `hr:hasAddress`. On a textual element, the address value is in the content; on an abbreviation, the expanded form (the *title* of the abbreviation) specifies the address; and on a hyperlink, the target of the link specifies the address of the operation.

The **`method`** class on textual markup (e.g. `<span>`, shown on line 5) specifies the HTTP method used by the operation. Its value is attached to the operation using the property `hr:hasMethod`.

Both the address and the method may also be specified on the level of the service, in which case these values serve as defaults for operations that do not specify them. In absence of any explicit value for the method, the default is GET. The RDF form of the service model reflects the default values already applied, that is, an instance `hr:Service` will never have either `hr:hasMethod` or `hr:hasAddress`.

The **`input`** and **`output`** classes are used on block markup (e.g. `<div>` but also `<span>`), as shown on lines 7 and 13, to indicate the description of the input or output of an operation. Elements with these classes correspond to instances of `hr:Message`, attached to the parent operation with `hr:hasInputMessage` and `hr:hasOutputMessage` respectively.

While the output data format can, in principle, be self-describing through the metadata the client receives together with the operation response, but it is, in general, useful for API descriptions to specify what the client can expect; hence the `output` class.

The class **`parameter`**, an extension of the original hRESTS microformat, is used on block markup as shown on line 9 to mark the description of a particular parameter of an input or output message. The class can be complemented with the class **`mandatory`** to indicate that the parameter is mandatory; otherwise it can be treated as optional. Elements with the `parameter` class correspond to instances of `hr:MessagePart`, attached to the parent input or output with `hr:hasMandatoryPart` or `hr:hasOptionalPart`, depending on the presence of the class `mandatory` on the element.

The **`label`** class is used on textual markup to specify human-readable labels for services, operations, messages and their parameters, as shown on lines 2 and 4 in the example listing. The value is attached to the appropriate service or operation using `rdfs:label`.

Additionally, service, operation, message and parameter elements can carry an **`id`** attribute, which is combined with the URI of the HTML document to form the URI identifier of the particular instance. This will allow other semantic statements to refer to these instances directly.

The definitions above imply a hierarchical use of the classes within the element structure of the HTML documentation. The following is a complete list of structural constraints on the hierarchy of elements marked up with hRESTS classes. It reflects the structure of our service model, amended with the defaulting of the `address` and `method` properties:

1. No XHTML element with the class `service` is a descendant[6] of an element with any hRESTS class.

[6]The term *descendant* is defined for XML/HTML elements in XPath (XML Path Language XPath 2009).

2. Every element with the class `operation` is a descendant of an element with the class `service`. No element with the class `operation` is a descendant of an element with an hRESTS class other than `service`.
3. Every element with the class `address` or `method` is a descendant of an element with either the class `service` or the class `operation`.
4. Every element with the class `input` or `output` is a descendant of an element with the class `operation`. Among the descendants of any given element with the class `operation`, there is no more than one element with the class `input` and no more than one element with the class `output`.
5. No element with any of the classes `address`, `method`, `input`, or `output` is a descendant of an element with an hRESTS class other than `service` and `operation`.
6. Every element with the class `parameter` is a descendant of an element with either the class `input` or the class `output`. No element with the class `parameter` is a descendant of an element with an hRESTS class other than `service`, `operation`, `input` and `output`.
7. No element with the class `label` is a descendant of an element an hRESTS class other than `service`, `operation`, `input`, `output`, `parameter` or `mandatory`.
8. No element has two or more hRESTS classes other than `mandatory` at the same time. The class `mandatory` is only permitted on elements with the hRESTS class `parameter`.

Microformat for SAWSDL

In "Minimal Service Model" (page 480), we have discussed the use of SAWSDL properties to add semantic annotations to service descriptions. Here, we define a simple microformat that extends hRESTS to support such semantic annotations.

SAWSDL annotations are URIs that identify semantic concepts and data transformations. Such URIs can be added to the HTML documentation of RESTful services in the form of hypertext links. HTML (HTML 4.01 Specification 1999) defines a mechanism for specifying the relation represented by link, embodied in the `rel` attribute; along with `class`, this attribute is also used to express microformats. In accordance with SAWSDL, we introduce the following three new types of link relations:

- `Model` indicates that the link is a model reference.
- `Lifting` and `lowering` denote links to the respective data transformations.

Listing 22.4 illustrates the use of these link relations on semantic annotations added to the hRESTS description from Listing 22.3. In the detailed definitions below, we refer to the SAWSDL RDF properties using the prefix `sawsdl`.[7]

[7]The prefix `sawsdl` refers to the namespace http://www.w3.org/ns/sawsdl#.

```
1  <div class="service" id="svc">
2   <h1><span class="label">ACME Hotels</span> service API</h1>
3    <p>This service is a
4     <a rel="model" href="http://example.com/ecommerce/hotelReservation">
5       hotel reservation</a> service.
6    </p>
7   <div class="operation" id="op1">
8    <h2>Operation <code class="label">getHotelDetails</code></h2>
9    <p> Invoked using the <span class="method">GET</span>
10    at <code class="address">http://example.com/h/{id}</code><br/>
11    <span class="input">
12     <strong>Parameters:</strong>
13     <span class="parameter mandatory">
14      <code class="label">id</code> – the identifier of the particular
15      <a rel="model" href="http://example.com/data/onto.owl#Hotel">hotel</a>
16     </span>
17     (<a rel="lowering" href="http://example.com/data/hotel.xsparql">lowering</a
        >)
18    </span><br/>
19    <span class="output">
20     <strong>Output value:</strong> hotel details in an
21     <code>ex:hotelInformation</code> document
22    </span>
23   </p>
24 </div></div>
```

Listing 22.4 Example hRESTS and SAWSDL semantic description

The **`model`** link relation, on a hyperlink present within an hRESTS `service`, `operation`, `input`, `output` or `parameter` block, specifies a model reference (`sawsdl:modelReference`) from the respective component to its semantic description.

Listing 22.4 shows the use of the `model` link relation on lines 4 and 15. Line 4 specifies that the service does hotel reservations (the URI would identify a category in some classification of services), whereas line 15 defines the input parameter of the operation to be an instance of the class Hotel, which is a part of the data ontology of this service.

The **`lifting`** and **`lowering`** link relations, on hyperlinks present within an hRESTS `input` or `output` block correspond with the properties `sawsdl:liftingSchemaMapping` and `sawsdl:loweringSchemaMapping`; they specify the respective data transformations between the knowledge representation format of the client and the syntax of the wire messages of the service.

Listing 22.4 shows a link to a lowering transformation on line 17. The transformation would presumably map a given instance of the class Hotel into the ID that the service expects as a URI parameter. The description of concrete data lifting and lowering technologies is out of scope of this chapter.[8]

[8]A notable new technology for transformations between XML and RDF (either way) is XSPARQL (Akhtar et al. 2008), see http://xsparql.deri.org.

```
1   @prefix hr:       <http://www.wsmo.org/ns/hrests#> .
2   @prefix rdfs:     <http://www.w3.org/2000/01/rdf−schema#> .
3   @prefix sawsdl:  <http://www.w3.org/ns/sawsdl#> .
4   @prefix ex:       <http://example.com/api/desc.html#> .
5
6   ex:svc  a  hr:Service ;
7     rdfs:isDefinedBy <http://example.com/api/desc.html> ;
8     rdfs:label  ”ACME Hotels” ;
9     sawsdl:modelReference  <http://example.com/ecommerce/hotelReservation> ;
10    hr:hasOperation  ex:op1 .
11  ex:op1  a  hr:Operation;
12    rdfs:label  ”getHotelDetails” ;
13    hr:hasMethod  ”GET” ;
14    hr:hasAddress  ”http://example.com/h/{id}”^^hr:URITemplate ;
15    hr:hasInputMessage  [
16      a  hr:Message ;
17      hr:hasMandatoryPart  [
18        a  hr:MessagePart ;
19        rdfs:label  ”id” ;
20        sawsdl:modelReference <http://example.com/data/onto.owl#Hotel>
21      ] ;
22      sawsdl:loweringSchemaMapping  <http://example.com/data/hotel.xsparql>
23    ] ;
24    hr:hasOutputMessage  [  a  hr:Message  ] .
```

Listing 22.5 RDF data extracted from Listing 22.4

Parsing hRESTS

As a microformat backed by an RDFS model, hRESTS can be processed by a parser to extract RDF data from HTML pages annotated with the microformat's classes. For example, there is an openly available XSLT stylesheet[9] that implements such a parser.

Listing 22.5 shows the RDF view of the hRESTS+SAWSDL description from Listing 22.4, assuming the description is located at `http://example.com/api/desc.html`. Most of the listing is self-explanatory (for readers familiar with the Turtle RDF syntax[10]). Note that the parser adds the `rdfs:isDefinedBy` property (line 7) on the service instance with a pointer back to the HTML documentation that defines it.

Service Description with the Minimal Service Model and RDFa

Alternative to using the hRESTS microformat to capture the service model structure in the HTML documentation of RESTful Web services, we can also employ RDFa (RDFa in XHTML: Syntax and Processing 2008) in order to use the RDF

[9]http://cms-wg.sti2.org/TR/d12/v0.1/20081202/xslt/hrests.xslt.

[10]http://www.w3.org/TeamSubmission/turtle/.

```
<div typeof="hr:Service" about="#svc"
     xmlns:hr="http://www.wsmo.org/ns/hrests#"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
 <span rel="rdfs:isDefinedBy" resource="" />
 <h1><span property="rdfs:label">ACME Hotels</span> service API</h1>
 <div rel="hr:hasOperation"><div typeof="hr:Operation" about="#op1">
  <h2>Operation <code property="rdfs:label">getHotelDetails</code></h2>
  <p>Invoked using the method <span property="hr:hasMethod">GET</span>
   at <code property="hr:hasAddress" datatype="hr:URITemplate"
          >http://example.com/h/{id}</code><br/>
   <span rel="hr:hasInputMessage"><span typeof="hr:Message">
    <strong>Parameters:</strong>
    <span rel="hr:hasMandatoryPart"><span typeof="hr:MessagePart">
     <code property="rdfs:label">id</code>
     – the identifier of the particular hotel</span></span>
   </span></span><br/>
   <span rel="hr:hasOutputMessage"><span typeof="hr:Message">
    <strong>Output value:</strong> hotel details in an
    <code>ex:hotelInformation</code> document
   </span></span>
  </p>
</div></div></div>
```

Listing 22.6 Example service description with RDFa annotations

service model directly. RDFa specifies a collection of generic XML attributes for expressing RDF data in any markup language, and especially in HTML.

Since our service description data is ultimately processed as RDF, RDFa is directly applicable. In our case, the difference between the use of a microformat or RDFa boils down to several considerations:

- The microformat syntax is simpler and more compact than RDFa;
- HTML marked up with our microformat remains valid HTML, whereas RDFa currently only validates against the newest schemas;
- RDFa represents the full concept URIs and thus facilitates the coexistence of multiple data vocabularies in a single document, where microformats may run into naming conflicts;
- Processing microformats requires vocabulary-specific parsers (such as our XSLT transformation described in "Parsing hRESTS" (page 488)), while parsing the RDF data from RDFa is independent from any actual data vocabularies.

To illustrate the RDFa annotations explained in the following text, Listing 22.6 shows the HTML description from Listing 22.1, annotated with RDFa as data in the minimal service model; all the extra markup is highlighted in bold.

First, any portion of the HTML document that describes a given part of the service (an operation, its input our output, or the service as a whole) should be enclosed in a single HTML container element, such as `<body>`, `<p>`, or in a general-purpose block such as `<div>` or `<span>`. In many cases this will already be so; otherwise the annotator can introduce a new container element with minimal effect on the presentation of the HTML document in a Web browser. In the listing, the added container elements are on lines 1–22 (service), 6–22 (operation), 11–16 (input), and 17–20 (output).

These container elements can then be marked with the RDFa `typeof` attribute with the appropriate service model class: `hr:Service` (line 1), `hr:Operation` (line 6), `hr:Message` (lines 11 and 17), or `hr:MessagePart` (line 13). To link a service to its operations, and the operations to their input and output messages, we use the RDF properties `hr:hasOperation`, `hr:hasInputMessage` and `hr:hasOutputMessage` in the RDFa `rel` attribute, as shown on lines 6, 11 and 17. To link message parts to their parent message, we use the RDF properties `hr:hasMandatoryPart` (line 13) or `hr:hasOptionalPart`, as appropriate.

The duplicate `<div>` and `<span>` container elements on lines 6, 11, 13 and 17 are required to explicitly type operations and messages with the respective service model classes. This type information can also be inferred from the RDFS schema of the service model, so the `typeof` annotations (and their associated extra container elements) could potentially be omitted.

Most of the components can also usefully carry human-readable labels, using the RDFS property `rdfs:label` in the RDFa `property` attribute used for textual properties (see lines 5, 7 and 14; note how identifying the label on line 5 also needed a `<span>` wrapper element). Additionally, as shown on line 4, the documentation should include an `rdfs:isDefinedBy` link to `""` (which means the HTML document itself).

A description of an operation can specify the URI template and the method where the operation can be invoked; for this, we use the properties `hr:hasAddress` and `hr:hasMethod` (shown on lines 8 and 9; identifying the method in this example needed another `<span>` wrapper).

Listing 22.7 shows the RDF data encoded in Listing 22.6, if that document were available at `http://example.com/api/desc.html`.

SAWSDL in RDFa

Like in hRESTS in the previous section, SAWSDL semantic annotations can be added to the HTML documentation of RESTful services in the form of hypertext links. In RDFa, they are marked with `sawsdl:modelReference`, `sawsdl:liftingSchemaMapping` or `sawsdl:loweringSchemaMapping` as the value of the RDFa `rel` attribute, as appropriate. If a clickable link is not appropriate for the presentation of a particular semantic annotation, some markup element other than `<a>` can be used, for example an empty `<span>` with the RDFa attributes `rel` and `resource`.

To illustrate SAWSDL semantic annotations on top of the example in Listing 22.6, Listing 22.8 contains two model references into an ontology at `http://example.com/onto.owl` and one link to a lowering mapping; all the extra markup is highlighted in bold.

```
@prefix hr: <http://www.wsmo.org/ns/hrests#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf−schema#> .
@prefix ex: <http://example.com/api/desc.html#> .

ex:svc  a  hr:Service ;
  rdfs:label  ”ACME Hotels” ;
  rdfs:isDefinedBy  <http://example.com/api/desc.html> ;
  hr:hasOperation  ex:op1 .
ex:op1  a  hr:Operation ;
  rdfs:label  ”getHotelDetails” ;
  hr:hasAddress  ”http://example.com/h/{id}”^^hr:URITemplate ;
  hr:hasMethod  ”GET” ;
  hr:hasInputMessage  [
    a  hr:Message ;
    hr:hasMandatoryPart  [
      a  hr:MessagePart ;
      rdfs:label  ”id”
    ]
  ] ;
  hr:hasOutputMessage  [  a  hr:Message  ] .
```

Listing 22.7 RDF data encoded in Listing 22.6 (in Turtle syntax)

```
<div typeof=”hr:Service” about=”#svc”
     xmlns:hr=”http://www.wsmo.org/ns/hrests#”
     xmlns:sawsdl="http://www.w3.org/ns/sawsdl#"
     xmlns:rdfs=”http://www.w3.org/2000/01/rdf−schema#”>
 <span rel=”rdfs:isDefinedBy” resource=”” />
 <h1><span property=”rdfs:label”>ACME Hotels</span> service API</h1>
  <p>This service is a
   <a rel="sawsdl:modelReference"
      href="http://example.com/ecommerce/hotelReservation">
    hotel reservation</a> service.
  </p>
 <div rel=”hr:hasOperation”><div typeof=”hr:Operation” about=”#op1”>
  <h2>Operation <code property=”rdfs:label”>getHotelDetails</code></h2>
  <p>Invoked using the method <span property=”hr:hasMethod”>GET</span>
   at <code property=”hr:hasAddress” datatype=”hr:URITemplate”
          >http://example.com/h/{id}</code><br/>
   <span rel=”hr:hasInputMessage”><span typeof=”hr:Message”>
    <strong>Parameters:</strong>
    <span rel=”hr:hasMandatoryPart”><span typeof=”hr:MessagePart”>
     <code property=”rdfs:label”>id</code>
     – the identifier of the particular
     <a rel="sawsdl:modelReference" href="http://example.com/onto.owl#Hotel">
      hotel</a>
    </span></span>
    (<a rel="sawsdl:loweringSchemaMapping"
      href="http://example.com/hotel.xsparql">lowering</a>)
   </span></span><br/>
   <span rel=”hr:hasOutputMessage”><span typeof=”hr:Message”>
    <strong>Output value:</strong> hotel details in an
    <code>ex:hotelInformation</code> document
   </span></span>
  </p>
</div></div></div>
```

Listing 22.8 Example RDFa service description extended with SAWSDL annotations

```
@prefix hr: <http://www.wsmo.org/ns/hrests#> .
@prefix sawsdl: <http://www.w3.org/ns/sawsdl#> .
@prefix ex: <http://example.com/api/desc.html#> .

ex:svc
   sawsdl:modelReference <http://example.com/ecommerce/hotelReservation> .

ex:op1
  hr:hasInputMessage [
    a hr:Message ;
    sawsdl:loweringSchemaMapping <http://example.com/hotel.xsparql> ;
    hr:hasMandatoryPart [
      a hr:MessagePart ;
      sawsdl:modelReference <http://example.com/onto.owl#Hotel>
    ]
  ] .
```

Listing 22.9 Additional RDF data encoded in Listing 22.8 (showing only differences from Listing 22.7; in Turtle syntax)

In the listing, the new paragraph on lines 7–11 contains a model reference that associates the service with the concept HotelReservation, which may be in a taxonomy of service functionalities; and the new link on line 22 represents a model reference that defines the `id` parameter to be an instance of the class Hotel.

Lines 24–25 show a link to a lowering transformation. The transformation would presumably map a given instance of the class Hotel into the ID that the service expects as a URI parameter.

The SAWSDL properties added in Listing 22.8 are shown in RDF in Listing 22.9. Note that combined with the contents of Listing 22.7, the RDF data is the same as that extracted from the microformat version in Listing 22.5.

Notably, an earlier work called SA-REST (Sheth et al. 2007) also used RDFa to express semantic annotations of services. In contrast with our work presented in this chapter, SA-REST used a limited implied service model, describing a single HTTP method on any given resource as an operation, without grouping operations into services.

Service Semantics with WSMO-Lite

In this section, we briefly describe how the lightweight service descriptions can be used to support automation of the use of RESTful Web services. The aim is to use semantic technologies to help with the following tasks: *discovery* matches known Web services against a user goal and returns the services that can satisfy that goal; *composition* puts together multiple services when no single service can fulfill the whole goal; *ranking* orders the discovered or composed services based on user requirements and preferences so the best service can be selected; *invocation* then communicates with the service to execute its functionality; and *mediation* resolves

```
@prefix rdfs:  <http://www.w3.org/2000/01/rdf−schema#> .
@prefix owl:   <http://www.w3.org/2002/07/owl#> .
@prefix wsl:   <http://www.wsmo.org/ns/wsmo−lite#> .

wsl:Ontology a rdfs:Class ;
    rdfs:subClassOf owl:Ontology .
wsl:FunctionalClassificationRoot rdfs:subClassOf rdfs:Class .
wsl:NonfunctionalParameter a rdfs:Class .
wsl:Precondition a rdfs:Class .
wsl:Effect a rdfs:Class .
```

Listing 22.10 WSMO-Lite ontology for service semantics

any arising heterogeneities. A semantic software system that automates these tasks acts on behalf of the actual user as a client to the services.

To support such automation, service descriptions need to capture four aspects of service semantics:[11] *information model* (a domain ontology) represents data, especially in input and output messages; *functional semantics* specifies what the service does; *behavioral semantics* defines the sequencing of operation invocations when invoking the service; and *nonfunctional descriptions* represent service policies or other details specific to the implementation or running environment of a service.

WSMO-Lite (Fensel et al. 2010) proposes a lightweight ontology for the four kinds of semantics, shown in Listing 22.10, intended to be combined with SAWSDL to annotate service descriptions. Informally, the four types of service semantics are represented in the WSMO-Lite ontology as follows:

- Information semantics are represented using domain *ontologies*, which are also involved in the descriptions of the other types of semantics.
- Functional semantics are represented in WSMO-Lite as *capabilities* and/or functionality *classifications*. A capability defines *preconditions* which must hold in a state before the client can invoke the service, and *effects* which hold in a state after the service invocation. Functionality classifications define the service functionality using some classification taxonomy (i.e., a hierarchy of *categories*).[12]
- Nonfunctional semantics are represented using an ontology that semantically captures some policy or other nonfunctional properties.
- Behavioral semantics are represented by annotating the service operations with functional descriptions, i.e., capabilities and/or functionality classifications. Functional annotations of operations can then serve for ordering of operation invocations.

[11]The separation of four types of service semantics is inspired by Sheth (2003).

[12]The distinction of capabilities and categories is the same that is made by Sycara et al. (2003) between "explicit capability representation" (using taxonomies) and "implicit capability representation" through preconditions and effects.

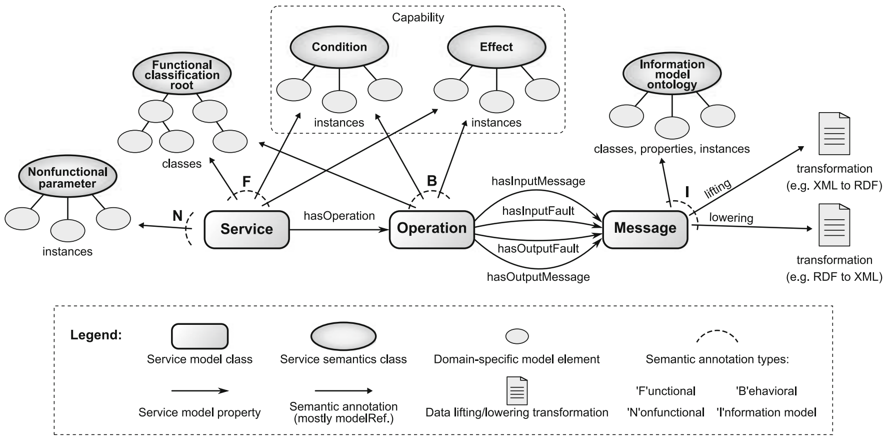


Fig. 22.5 The structure and use of the WSMO-Lite service ontology, annotating the elements of the minimal service model

Functional and nonfunctional semantics are directly properties of a service. Behavioral semantics tie to service operations. Finally, information semantics tie to the data that a service communicates with – to the input, output and fault messages of the operations. Figure 22.5 illustrates the WSMO-Lite annotations in relation to the service model. The centrally-located components of the service model are annotated with pointers to domain-specific semantic descriptions that fit the service semantics classes defined in WSMO-Lite.

In the interest of simplicity of the RDF form of actual concrete semantic service descriptions, the ontology is not a straightforward implementation of the formal terms (such as *classification*, *capability*, or *ontology for nonfunctional semantics*). We discuss below some of the considerations that led to the presented form of the ontology classes.

`wsl:Ontology` is a class that serves to mark an information model ontology. Similarly to `owl:Ontology` from the standard Web Ontology Language OWL (OWL Web Ontology Language Overview 2004), `wsl:Ontology` allows for meta-data such as comments, version control and inclusion of other ontologies. `wsl:Ontology` is a subclass of `owl:Ontology`, restricted only to ontologies intended to capture a service information model, as opposed to other kinds of ontologies.

The class `wsl:Ontology` can be used by tools for authoring semantic service descriptions, for instance to primarily suggest explicitly-marked information ontologies when annotating data schemas.

`wsl:FunctionalClassificationRoot` is a class that marks the roots of service functionality classifications. In other words, for every functionality taxonomy, the root class of the taxonomy is an instance of this class. All subclasses of the root are included in the particular classification. An annotation tool can simply suggest all functional classification root classes and their subclasses when creating functional annotations.

`wsl:NonfunctionalParameter` is a class of concrete, domain-specific nonfunctional parameters. For a particular ontology of nonfunctional semantics, its instances would be instances of this class.[13]

`wsl:Condition, wsl:Effect` together form a *capability* in a functional service description. Instances of these classes are expected to contain some logical expressions. The WSMO-Lite service ontology does not specify the concrete language for the logical expressions, or their processing. Logical expressions can be specified in any suitable language, such as RIF (RIF Core Dialect 2010), SWRL (Horrocks 2003), or WSML (The Web Service Modeling Language WSML 2008), and embedded in RDF semantic descriptions as literals.

Now that we have shown the ontology for expressing semantic service descriptions, we briefly discuss several algorithms that can be used by semantic client software to process WSMO-Lite descriptions to automate some of the tasks involved in the use of Web services.

The process of using Web services can be split into the following tasks: discovery, negotiation, ranking and selection, composition, mediation and invocation. Algorithms for automating these tasks have been researched in the area of Semantic Web Services (SWS); they can commonly be adapted to WSMO-Lite with little effort.

Since WSMO-Lite semantics is intentionally lightweight, adapting a SWS automation algorithm may involve filling in concrete details about WSMO-Lite semantics that are used by the algorithm, effectively refining the semantics defined by WSMO-Lite. Additionally, an algorithm must also define what kinds of data it requires from the user to specify a *goal*. While this means that different algorithms for the same automation task need not be able to process the same semantic service and goal descriptions, WSMO-Lite aims to provide a limited common ground for the various approaches to semantic automation, to facilitate communication between the often disconnected SWS research groups.

In the rest of this section, we discuss possible algorithms for a few selected tasks, demonstrating both the use of WSMO-Lite annotations and the refinement of its semantics.

[13]WSMO-Lite does not place any further restrictions on nonfunctional parameters; research in this area, which is out of scope of this book, has not yet converged on a common set of properties that nonfunctional parameters should have.

Functional Service Discovery

For discovery (also known as "matchmaking") purposes, WSMO-Lite provides functional service semantics of two forms: functionality classifications and precondition/effect capabilities, with differing discovery algorithms.

With functionality classifications, a service is annotated with particular functionality categories (e.g. *HotelReservation*, shown in the examples earlier in this section). We treat the service as an instance of these category classes. The user goal will identify a concrete category of services that the user needs (let's say *AccommodationReservation*). A discovery mechanism uses subsumption reasoning among the functionality categories to identify the services that are members of the goal category class ("direct matches"). If no such services are found, a discovery mechanism may also identify instances of progressively further superclasses of the goal category in the subclass hierarchy of the functionality classification. To illustrate: if the user is looking for a *AccommodationReservation*, it will find services marked as *HotelReservation* (presuming the intuitive subclass relationships) as direct matches, and it may find services marked as a more generic *TourismService* which also potentially do accommodation reservation, even though the description does not directly advertise that.

For discovery with preconditions and effects, the user goal must specify the requested effects. The discovery mechanism will need to check, for every available service, that the user's knowledge base fulfills the precondition of the service, and that the effect of the service fulfills the effect requested by the user. This is achieved using satisfaction and entailment reasoning.

Discovery using functionality categorizations is likely to be coarse-grained, whereas the detailed discovery using preconditions and effects may be complicated for the users and resource-intensive. Therefore, the two approaches should be combined to describe the core functionality in general classifications, and only some specific details using logical expressions, resulting in better overall usability.

Service Filtering, Ranking and Selection

These tasks mostly deal with the nonfunctional parameters of a service. The user goal (or general user settings) specifies constraints and preferences (also known as hard and soft requirements) on a number of different aspects of the discovered services and offers. For instance, service price, availability and reliability are typical parameters for services.

Filtering is implemented simply by comparing user constraints with each service's parameter values, resulting in a binary (yes/no) decision. Ranking, however, is a multidimensional optimization problem, and there are many approaches to dealing with it, including aggregation of all the dimensions through weighted preferences into a single metric by which the services are ordered, or finding locally-optimal services using techniques such as Skyline Queries (Skoutas et al. 2008).

Selection is then the task of selecting only one of the ranked services. With a total order, the first service can be selected automatically, but due to the complexity of comparing the different nonfunctional properties (for instance, is a longer warranty worth the higher price?), often the ordered list of services will be presented to the user for manual selection.

Service Composition

Service composition is the process of combining existing services in such a way that they provide a new desired functionality; the result is also often called a service composition, or a composite service. A composition may simply be a linear sequence of services, or it can be a non-linear process with parallel and/or conditional branches.

Some composition approaches match services into a sequence based on their inputs and outputs, assuming that the inputs and outputs of a service implicitly reflect the service's functionality. More sophisticated approaches use the preconditions and effects of Web services as *explicit* functional descriptions, decoupling message types from service functionality.

Hoffmann et al. (2008) provide an example of a powerful composition algorithm that deals with service preconditions and effects. Overall, it is a typical state-space search algorithm that searches in a space of beliefs representing what happens after applying various available services. The initial belief combines a background ontology with the data provided by the user's goal. The search is successful when it finds a belief that satisfies the effect requested in the user's goal; the sequence of services that leads to the belief is then a solution of the composition algorithm.

Creating new models that describe the effect of service invocations is called *update reasoning*, and it is a known hard problem (Hoffmann et al. 2008). uses tractable approximate reasoning with Horn theories: the algorithm computes an under-approximation and an over-approximation of the statements that hold after applying a service. A solution is guaranteed if the goal effect is satisfied in the under-approximated view on the current belief, and a solution is only potentially found if the goal effect is satisfied in the over-approximated belief. In an iterative system, the search algorithm can present to the user all the potential solutions it encounters while continuing to search for a guaranteed solution, increasing the responsiveness of the user interface.

After the algorithm finds a composition solution, whether a potential one found with the over-approximated reasoning, or a solution guaranteed by the under-approximated reasoning, the composition can be presented to the user, who may need to fill in details of data or process mediation. If multiple potential solutions are found, it can be useful to rank the compositions according to the combined nonfunctional properties of their constituent services.

Tools and Implementations

In “hRESTS: Microformat for Service Descriptions” (page 483), we have already mentioned one implementation related to the lightweight languages presented in this chapter: a parser for the hRESTS microformat. The RDFa alternative to annotating service documentation also has many parsers available.[14]

Above parsers, there are further types of implementations that can process the lightweight service descriptions. In this section, we describe two software systems: SWEET, an editor and semantic annotator for service descriptions of Web APIs; and iServe, a semantic service registry. In general, the following are some of the main types of implementations and tools that we can envision:

- *Service description editors and annotators:* whether starting from scratch to document a new service, or structuring the documentation of an existing service, the user can be supported by an editing tool to follow the service model presented in this chapter. Similarly, a tool can help a user with adding semantic annotations to service descriptions. Describing Web services semantically is a knowledge-intensive task that cannot be fully automated without strong artificial intelligence, but the task can be eased by suggesting appropriate ontologies and by guiding the user in applying semantic annotations to the underlying Web service descriptions. SWEET is one example of an editor capable of adding the minimal service model structure to existing HTML files and recommending and adding SAWSDL semantic annotations.
- *Service registries and search engines:* machine-processable service descriptions are only useful if they are available to clients. Public and private registries can gather service descriptions and facilitate search over them, whether semantic or not. iServe is such a registry built for the minimal service model and its semantic annotations.
- *Discovery, ranking and composition:* these semantic algorithms are best implemented in conjunction with a service registry that provides the descriptions of known services. As described below, iServe includes rich discovery functionality, and tools for ranking and composition are in development.
- *Client-side code generation:* with a machine-processable service description for a RESTful API, some client-side code for accessing the API can be generated, hiding away the details of how parameters are put in the request message, what HTTP method corresponds to what operation, and the low-level code that drives the HTTP request/response interaction. For RESTful services, such tools currently only exist for WADL, but the lightweight service descriptions that use the minimal service model can also support such tools.
- *Invocation:* to invoke a RESTful API, whether on direct input from a user, or as a part of a composition, there must be a system capable of executing the lifting and lowering data transformations, acting as middleware between the semantic client and the semantically-described services. Currently, an invocation platform is under development.

[14]http://rdfa.info/rdfa-implementations/.

A WSMO-Lite-based semantic service environment, SOA4All Studio, is being developed in the research project SOA4All.[15] When finished, it will contain all the above tools, including SWEET and iServe.

iServe: A Service Registry

iServe is a public platform[16] that unifies service publication and discovery on the Web through the use of lightweight semantics. The service descriptions published in iServe are avilable on the Web as RDF, following the principles of Linked Data (Bizer et al. 2009):

1. Use URIs as names for things.
2. Use HTTP URIs so that people can look up those names.
3. When someone looks up a URI, provide useful information, using the standards (RDF*, SPARQL).
4. Include links to other URIs, so that they can discover more things.

iServe provides a generic semantic service registry able to support advanced discovery over different kinds of services described using heterogeneous formalisms. The fundamental objective pursued by iServe is to provide a platform able to publish service annotations in a way that would allow people to achieve a certain level of expressivity and refinement when discovering services, while remaining simple and convenient both for human and machine usage.

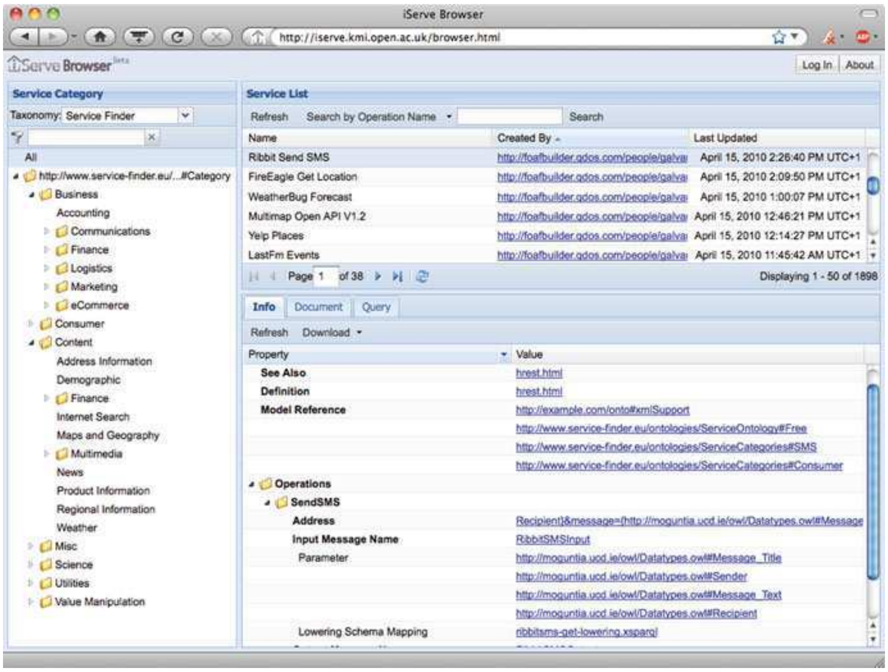
Currently iServe provides import support for hRESTS with SAWSDL,[17] WSDL with SAWSDL, and OWL-S (OWL-S 1.1 Release 2004). The mapping from OWL-S is not lossless, nor is it meant to be. Instead, it extracts the “bare bones” of the original description that are compatible with the minimal service model, keeping an `rdfs:isDefinedBy` link to the original description. This way iServe publishes service descriptions in the Web of Data in a common way that is amenable to automated processing; systems optimized for specific formalisms can still obtain and exploit the original descriptions.

The main components of iServe are a browser GUI, a set of RESTful APIs, a crawler, a set of import mechanisms, and an RDF store. The iServe browser GUI, shown in Fig. 22.6 is a Web-browser-based application that is the main human-oriented interface of the registry. The RESTful APIs provide support for accessing and submitting service annotations and service documentation, along with several types of semantic discovery, as detailed below. The crawler takes care of collecting existing annotations from the Web in order to publish them in iServe. Using the

[15]http://soa4all.eu.

[16]Located at http://iserve.kmi.open.ac.uk/.

[17]iServe import support for the RDFa form described in “Service Description with the Minimal Service Model and RDFa” (page 488) is planned.



Taxonomy: Service Finder

All
- http://www.service-finder.eu/...#Category
  - Business
    - Accounting
    - Communications
    - Finance
    - Logistics
    - Marketing
    - eCommerce
  - Consumer
  - Content
    - Address Information
    - Demographic
    - Finance
    - Internet Search
    - Maps and Geography
    - Multimedia
    - News
    - Product Information
    - Regional Information
    - Weather
  - Misc
  - Science
  - Utilities
  - Value Manipulation

Name	Created By	Last Updated
Ribbit Send SMS	http://foafbuilder.qdos.com/people/galva	April 15, 2010 2:26:40 PM UTC+1
FireEagle Get Location	http://foafbuilder.qdos.com/people/galva	April 15, 2010 2:09:50 PM UTC+1
WeatherBug Forecast	http://foafbuilder.qdos.com/people/galva	April 15, 2010 1:00:07 PM UTC+1
Multimap Open API V1.2	http://foafbuilder.qdos.com/people/galva	April 15, 2010 12:46:21 PM UTC+1
Yelp Places	http://foafbuilder.qdos.com/people/galva	April 15, 2010 12:14:27 PM UTC+1
LastFm Events	http://foafbuilder.qdos.com/people/galva	April 15, 2010 11:45:42 AM UTC+1

Page 1 of 38 Displaying 1 - 50 of 1898

Property	Value
See Also	hrest.html
Definition	hrest.html
Model Reference	http://example.com/onto#xmlSupport
	http://www.service-finder.eu/ontologies/ServiceOntology#Free
	http://www.service-finder.eu/ontologies/ServiceCategories#SMS
	http://www.service-finder.eu/ontologies/ServiceCategories#Consumer
Operations	
SendSMS	
Address	Recipient}&message={http://moguntia.ucd.ie/owl/Datatypes.owl#Message
Input Message Name	RibbitSMSInput
Parameter	http://moguntia.ucd.ie/owl/Datatypes.owl#Message_Title
	http://moguntia.ucd.ie/owl/Datatypes.owl#Sender
	http://moguntia.ucd.ie/owl/Datatypes.owl#Message_Text
	http://moguntia.ucd.ie/owl/Datatypes.owl#Recipient
Lowering Schema Mapping	ribbitsms-get-lowering.xsparql

Fig. 22.6 A screenshot of the iServe browser GUI, showing service categorizations on the *left*, a list of services in the *top-right* part, and the details of a selected service in the *bottom-right* part

crawler, iServe has imported several known large sets of service annotations, such as the SAWSDL and OWL-S test collections.[18] The import mechanisms process submitted descriptions in diverse formalisms by transforming them into the minimal service model. Finally, the RDF store manages all the service description and annotation data along with some provenance metadata; it also provides a SPARQL endpoint for advanced data access.

iServe Discovery API

In line with the general RESTfulness of iServe, discovery functionality is made available through a Web API; as such, it deserves a more detailed description here within this book. Currently, iServe implements three types of discovery: (1) discovery with functionality taxonomies, (2) matching of input and output signatures, and (3) similarity-based approximate matchmaking. The first two types only take into account direct logical relationships between semantic concepts,

[18] http://www.semwebcentral.org/projects/owls-tc/, http://www.semwebcentral.org/projects/sawsdl-tc/.

whereas the third uses information retrieval techniques that avoid strict logical false negatives.

The discovery mechanisms offered by iServe are available as part of the registry's RESTful API as follows:

`/data/disco/func-rdfs?class=`C_1`&class=`C_2`&...`
: uses RDFS functional classification annotations and returns those services that are related to all the functional categories C_i (which are URIs).

`/data/disco/io-rdfs?f={and|or}&i=`C_1^I`&i=`C_2^I`&o=`C_1^O`&...`
: uses ontology annotations of inputs and outputs and returns services for which the client has suitable input data (C_i^I) and/or (depending on the parameter `f` for *function*) which provide the outputs requested by the client (C_i^O).

`/data/disco/imatch?strategy=levenshtein&label=`L
: returns all services ranked according to string similarity of the service label with the string L.

In the spirit of using Web standards, the API represents discovery results as Atom feeds,[19] with the entries representing matching services sorted by matching degree. The Atom feed format was chosen for several reasons: it is a standard generic container format with wide support in software libraries and products, and it defines strong metadata properties (such as titles, identities and update times) that make feed readers a meaningful standalone software category. With Atom, iServe discovery queries can, for example, be syndicated and manipulated in generic systems such as Yahoo! Pipes,[20] or end users can watch for new interesting services by registering iServe discovery queries in their feed readers.

The common representation of discovery results as Atom feeds can be exploited for supporting arbitrary combinations of discovery approaches through list operations on the results of separate discovery queries. iServe includes three Atom feed combinators:

1. Union: the resulting feed contains the entries of all the constituent feeds. For discovery queries, the union of results is equivalent to the *or* (disjunction) operator: a service is returned if it matches any of the given queries.
2. Intersection: results in a feed with only the entries present in all the constituent feeds. This is equivalent to the *and* (conjunction) operator for discovery queries.
3. Subtraction: results in a feed with the entries of the first feed that are not in any other provided feed. In discovery, this enables the *and not* operator: it can return services that match one query but not another.

All these combinators are part of iServe's RESTful API, and they take feed URIs as parameters. To illustrate the use of the discovery API, including the Atom combinators, the following URI would discover proximity search services that take

[19] Atom Syndication Format, see http://rfc.net/rfc4287.html.

[20] http://pipes.yahoo.com.

as inputs a raw address (*proximity search* and *raw address* are terms in an ontology used to annotate a set of geography services present in iServe):

```
http://iserve.kmi.open.ac.uk/data/atom/intersection?
 f=/data/disco/func-rdfs?class=
  http://iserve.kmi.open.ac.uk/2010/05/s3eval/func.rdfs%2523ProximitySearch
&f=/data/disco/io-rdfs?
  i=http://iserve.kmi.open.ac.uk/2010/05/s3eval/data.rdfs%2523RawAddress
```

The example contains altogether five URIs: the location of the intersection combinator, the location of the RDFS functional classification discovery service (note that the URI is relative to the atom combinator URI), the identifier of a class of proximity search services, the location of the RDFS input/output matchmaker and the identifier of the concept of a raw address. Note how the nesting of URIs requires careful percent-encoding of special characters (such as the hash sign "#" encoded as "%23", which is then encoded again as "%2523" – "%25" is the percent-sign – because the URI is nested in two others).

The separation of the individual discovery algorithms from the mechanism by which they are combined supports easy extensibility: new discovery algorithms can be added to iServe independently (as plug-ins) and then usefully combined with the algorithms that are already there.

SWEET: Annotating Service Descriptions

SWEET is a Web-browser-based application[21] that supports the creation of semantic descriptions of Web APIs. SWEET takes as input an HTML Web page describing a Web API, and it allows the user to mark up the service structure and to annotate it with semantic information.

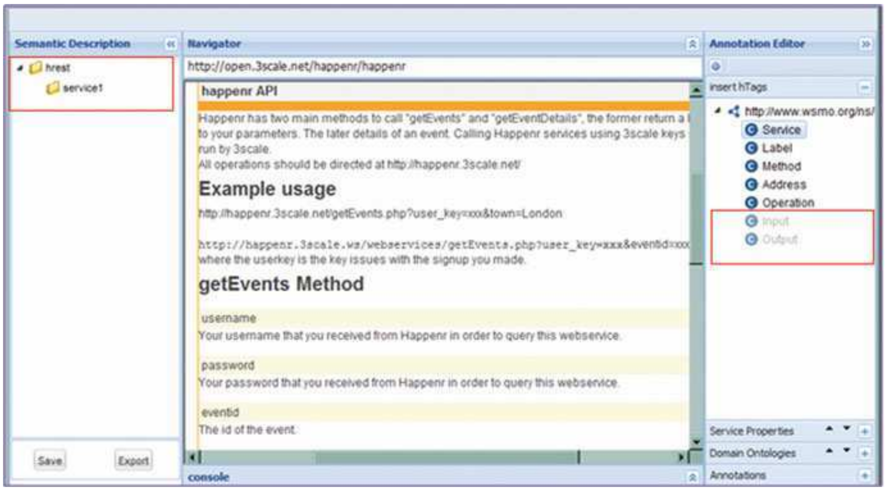
The tool is shown in Fig. 22.7. Its user interface has three main panels: the *Navigator* panel contains the HTML description of a selected Web API, to be used as a basis for the annotation process; the *Annotation Editor* panel contains allowed and recommended annotations, both for the hRESTS structure (shown), and for semantic properties; finally the *Semantic Description* panel visualizes the current service model structure of the documentation.

To annotate a selected Web API description, the user needs to complete the following four mains steps:

1. identify service and operation structure,
2. search for domain ontologies suitable for semantic annotations of this service,
3. annotate the service structure with semantic information,
4. save or export the annotated description.

For the first step, the user simply selects a part of the HTML that contains a particular chunk of the service model, and double-clicks on the corresponding tag

[21] Available at http://sweet.kmi.open.ac.uk/.



http://open.3scale.net/happenr/happenr

happenr API

Happenr has two main methods to call "getEvents" and "getEventDetails", the former return a
to your parameters. The later details of an event. Calling Happenr services using 3scale keys
run by 3scale.
All operations should be directed at http://happenr.3scale.net/

Example usage

http://happenr.3scale.net/getEvents.php?user_key=xxx&town=London

```
http://happenr.3scale.ws/webservices/getEvents.php?user_key=xxx&eventid=xxx
```
where the userkey is the key issues with the signup you made.

getEvents Method

username
Your username that you received from Happenr in order to query this webservice.

password
Your password that you received from Happenr in order to query this webservice.

eventid
The id of the event.

Fig. 22.7 Annotating a Web API description with SWEET

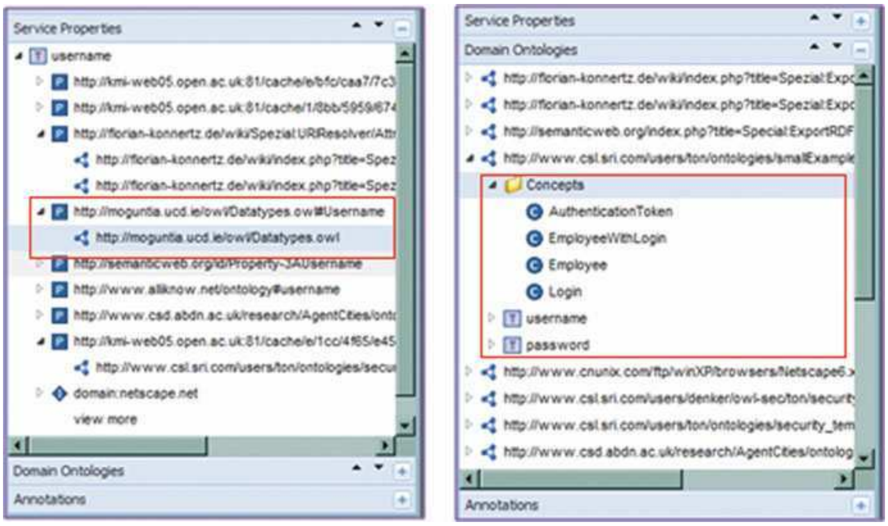
in the *insert hTags* pane. In the beginning, only the *Service* node of the hRESTS tree is enabled. After the user marks the body of the service description, additional tags are enabled. In this way, SWEET guides the user through the process of marking parts of the service description with hRESTS tags. The marking of HTML content with a particular hRESTS tag results in the insertion of a corresponding class HTML attribute, and is reflected in the *Semantic Description* panel. In addition, each inserted tag is highlighted to visualize the annotations.

When the HTML documentation is ready with a machine-processable hRESTS service model structure, the user can start adding semantic annotations. SWEET supports users in searching for suitable ontologies by providing an integrated search with Watson.[22] The user selects a part of the service description and the system then retrieves matching ontology entities from Watson, displaying them in the *Service Properties* and *Domain Ontologies* panels visualized in Fig. 22.8. Using the information in these panels, the user can choose a suitable ontology for annotating the API description.

By using Watson, SWEET assists users in locating appropriate annotations from among the existing ontological data on the Web, fostering ontology reuse.

Having chosen an appropriate ontology, the user can again pick parts of the service HTML description and add concrete semantic annotations through the context menu in the *Service Properties* panel. This results in inserting a model reference pointing to the URI of the selected semantic entity. The *Annotations* panel summarizes the already made annotations and makes it possible to delete them.

[22]Watson Semantic Web Search, http://watson.kmi.open.ac.uk.



```
Service Properties
  username
    http://kmi-web05.open.ac.uk:81/cache/e/bfc/caa7/7c3
    http://kmi-web05.open.ac.uk:81/cache/1/8bb/5959/674
    http://florian-konnertz.de/wiki/Spezial:URIResolver/Attr
      http://florian-konnertz.de/wiki/index.php?title=Spez
      http://florian-konnertz.de/wiki/index.php?title=Spez
    http://moguntia.ucd.ie/owl/Datatypes.owl#Username
      http://moguntia.ucd.ie/owl/Datatypes.owl
    http://semanticweb.org/id/Property-3AUsername
    http://www.alliknow.net/ontology#username
    http://www.csd.abdn.ac.uk/research/AgentCities/ont
    http://kmi-web05.open.ac.uk:81/cache/e/1cc/4f65/e45
      http://www.csl.sri.com/users/ton/ontologies/secu
    domain:netscape.net
    view more
Domain Ontologies
Annotations
```

```
Service Properties
Domain Ontologies
  http://florian-konnertz.de/wiki/index.php?title=Spezial:Expo
  http://florian-konnertz.de/wiki/index.php?title=Spezial:Expo
  http://semanticweb.org/index.php?title=Special:ExportRDF
  http://www.csl.sri.com/users/ton/ontologies/smallExample
    Concepts
      AuthenticationToken
      EmployeeWithLogin
      Employee
      Login
    username
    password
  http://www.cnunix.com/ftp/winXP/browsers/Netscape6.x
  http://www.csl.sri.com/users/denker/owl-sec/ton/securit
  http://www.csl.sri.com/users/ton/ontologies/security_tem
  http://www.csd.abdn.ac.uk/research/AgentCities/ontolog
Annotations
```

Fig. 22.8 Exploring domain ontologies in SWEET

When the user completes the semantic annotation of the HTML description, the result can be saved locally or it can be directly published in the iServe registry.

In summary, Web API providers or interested third parties can use SWEET as a user-friendly way of preparing machine-processable service descriptions, enabling tool support for discovery and so on. SWEET minimizes the effort involved in editing service descriptions, especially since it starts with the already-existing Web API documentation.

Summary

In this chapter, we have tackled the need for machine-processable service descriptions for RESTful services and Web APIs, in face of the reluctance of service providers to publish descriptions in languages such as WSDL and WADL.

The minimal service model presented in this chapter views services as sets of operations; while it disregards resources, we have argued that it is nevertheless a natural model for client-side tool support.

The service model can be applied to existing service documentation in HTML, using either the microformat hRESTS, or using RDFa, a generic extension of HTML for including RDF data. Either way, the human-oriented service documentation becomes a machine-processable service description with no repetition of information, and with minimal changes to the actual presentation of the documentation.

Having machine-processable service description is a prerequisite for adding semantic annotations that can support a degree of automation of service discovery and use. We have presented WSMO-Lite, a lightweight ontology for service semantics that embodies the spirit of minimizing the effort necessary for creating service descriptions. WSMO-Lite applies over the W3C standard SAWSDL equally well to RESTful services as it does to WSDL, and as such it puts aside the differences between the different Web service technologies, facilitating their interoperability via the level of semantics.

With the expected proliferation of Web APIs and other RESTful services, tool support and automation will grow in importance; lightweight service description approaches can lower the cost of adoption for service providers and speed up the emergence of a Web of Services.

Acknowledgments The authors would like to thank Dr. Karthik Gomadam for collaboration on the hRESTS microformat. Much of the work presented in this chapter has been supported by the European Union research project SOA4All http://www.soa4all.eu.

References

Akhtar, W., Kopecký, J., Krennwallner, T., Polleres, A.: XSPARQL: Traveling between the XML and RDF worlds – and avoiding the XSLT pilgrimage. In: S. Bechhofer, M. Koubarakis (eds.) The Semantic Web: Research and Applications, 5th European Semantic Web Conference, ESWC 2008, *Lecture Notes in Computer Science, LNCS*, vol. 5021, pp. 674–689. Springer, Tenerife, Spain (2008)

Akkiraju, R., Farrell, J., Miller, J., Nagarajan, M., Schmidt, M., Sheth, A., Verma, K.: Web Service Semantics – WSDL-S. Technical note (2005). Available at http://lsdis.cs.uga.edu/library/download/WSDL-S-V1.html

Architecture of the World Wide Web. Recommendation, W3C (2004). Available at http://www.w3.org/TR/webarch/

Bizer, C., Heath, T., Berners-Lee, T.: Linked Data - The Story So Far. International Journal on Semantic Web and Information Systems (IJSWIS), Special Issue on Linked Data (2009)

Fensel, D., Fischer, F., Kopecký, J., Krummenacher, R., Lambert, D., Vitvar, T.: WSMO-Lite: Lightweight Semantic Descriptions for Services on the Web (2010). URLhttp://www.w3.org/Submission/WSMO-Lite/. W3C member submission, available at http://www.w3.org/Submission/WSMO-Lite/

Gleaning Resource Descriptions from Dialects of Languages (GRDDL). Recommendation, W3C (2007). Available at http://www.w3.org/TR/grddl/

Hoffmann, J., Weber, I., Scicluna, J., Kaczmarek, T., Ankolekar, A.: Combining Scalability and Expressivity in the Automatic Composition of Semantic Web Services. In: Proceedings of the 8th International Conference on Web Engineering (ICWE'08). Yorktown Heights, USA (2008)

Horrocks, I., Patel-Schneider, P.F., Boley, H., Tabet, S., Grosof, B., Dean, M.: SWRL: A Semantic Web Rule Language Combining OWL and RuleML. Tech. rep., Joint US/EU ad hoc Agent Markup Language Committee (2003). Available at http://www.daml.org/2003/11/swrl/

HTML 4.01 Specification. Recommendation, W3C (1999). Available at http://www.w3.org/TR/html401

Khare, R., Çelik, T.: Microformats: a pragmatic path to the semantic web (Poster). Proceedings of the 15th international conference on World Wide Web pp. 865–866 (2006)

OWL Web Ontology Language Overview. Recommendation 10 February 2004, W3C (2004). Available at http://www.w3.org/TR/owl-features/

OWL-S 1.1 Release. Tech. rep., OWL Services Coalition (2004). Available at http://www.daml.org/services/owl-s/1.1/

RDFa in XHTML: Syntax and Processing. Recommendation, W3C (2008). Available at http://www.w3.org/TR/rdfa-syntax/

Richardson, L., Ruby, S.: RESTful Web Services. O'Reilly Media (2007)

RIF Core Dialect. Recommendation, W3C (2010). Available at http://www.w3.org/TR/rif-core/

Semantic Annotations for WSDL and XML Schema. Recommendation, W3C (2007). Available at http://www.w3.org/TR/sawsdl/

Sheth, A.P.: Semantic Web Process Lifecycle: Role of Semantics in Annotation, Discovery, Composition and Orchestration (2003). Invited Talk, Workshop on E-Services and the Semantic Web, at WWW 2003. Available at http://lsdis.cs.uga.edu/lib/presentations/WWW2003-ESSW-invitedTalk-Sheth.pdf

Sheth, A.P., Gomadam, K., Lathem, J.: SA-REST: Semantically Interoperable and Easier-to-Use Services and Mashups. IEEE Internet Computing **11**(6), 91–94 (2007)

Skoutas, D., Sacharidis, D., Simitsis, A., Sellis, T.: Serving the Sky: Discovering and Selecting Semantic Web Services through Dynamic Skyline Queries. In: Proceedings of the 2008 IEEE International Conference on Semantic Computing. Santa Clara, USA (2008)

Sycara, K., Paolucci, M., Ankolekar, A., Srinivasan, N.: Automated discovery, interaction and composition of Semantic Web services. Web Semantics: Science, Services and Agents on the World Wide Web **1**(1), 27–46 (2003)

The Web Service Modeling Language WSML. Tech. rep., WSMO Working Group (2008). Available at http://www.wsmo.org/TR/d16/d16.1/v1.0/

Web Services Description Language (WSDL) Version 2.0: Adjuncts. Recommendation, W3C (2007). Available at http://www.w3.org/TR/wsdl20-adjuncts/

XML Path Language (XPath) Version 1.0. Recommendation, W3C (2009). Available at http://www.w3.org/TR/xpath

Chapter 23
Towards Distributed Atomic Transactions over RESTful Services

Guy Pardon and Cesare Pautasso

Try-Cancel/Confirm: Transactions For the REST of Us

- Atomikos.com

Abstract There is considerable debate in the REST community whether or not transaction support is needed and possible. This chapter's contribution to this debate is threefold: we define a business case for transactions in REST based on the Try-Cancel/Confirm (TCC) pattern; we outline a very light-weight protocol that guarantees atomicity and recovery over distributed REST resources; and we discuss the inherent theoretical limitations of our approach. Our TCC for REST approach minimizes the assumptions made on the individual services that can be part of a transaction and does not require any extension to the HTTP protocol. A very simple but realistic example helps to illustrate the applicability of the approach.

Introduction

The Uniform Interface (Fielding 2000) of a RESTful Web service (Richardson and Ruby 2007) implemented using HTTP has very useful and positive implications on the reliability of the interaction of clients with a service following the constraint. Considering that `GET`, `PUT`, `DELETE` requests are by definition idempotent, any failure during these interactions can be addressed by simply repeating the request.

This property, however, cannot be directly applied in a service composition scenario (Pautasso 2009) where multiple interactions between a set of RESTful services need to happen atomically. Even if a single idempotent interaction between one client and one RESTful Web service is reliable, it is not clear how to guarantee the same property of atomicity when a client is interacting with multiple RESTful Web services. This problem is the central topic of this chapter, and will be discussed by means of the running example illustrated in the following section.

G. Pardon (✉)
ATOMIKOS, Hoveniersstraat 39/1, 2800 Mechelen, Belgium
e-mail: guy@atomikos.com

Example: Booking Two Connecting Flights

Suppose we want to book a flight composed of two connecting flights from two different and autonomous airlines: swiss.com and easyjet.com, via some travel agency service acting as a service composition over the two airlines. Let's assume that both airlines have the same hypermedia contract for bookings (for reasons of simplicity, and without loss of generality since the composite service is supposed to know all of the hypermedia contracts involved). The REST implementation of the airline information and booking services could be designed as follows.

Checking Seat Availability

Clients can inquire about the availability of seats on a flight at the URI: `/flight/{flight-no}/seat/{seat-no}`. For example, the `GET/flight /LX101/seat/` request will return a hyperlink to the next available seat on the flight `LX101` or none (e.g., `204 No Content`) if the flight is fully booked.

Booking a Seat

A `POST` request to the `/booking` URL with a payload referencing such seat will create a new booking resource and redirect the client to it by sending a hyperlink identifying it such as `/booking/{id}/`. The body of the request can contain a reference to the chosen flight and seat (i.e., `<flight number="LX101" seat="33F"/>`). The booking can be updated with additional information using a `PUT/booking/{id}/` request.

Composition of Bookings

We are now ready to present the first user story, which will be our motivating example throughout this chapter.

Story 1 *As a customer, I want to book a composed flight consisting of two independent, connecting flights from both airlines.*

It is the responsibility of the travel agency composite service to satisfy this requirement. A straightforward implementation (without a transaction model for REST) would be the following:

1. `GET swiss.com/flight/LX101/seat/`
2. `POST swiss.com/booking`
3. `GET easyjet.com/flight/EZ222/seat/`
4. `POST easyjet.com/booking`

The problem is that it may happen that after the first airline service has successfully performed the booking (step 2), the second airline may reply that there are no seats available. Thus we have only a partial flight.

Even if we reorder the requests as follows:

1. `GET swiss.com/flight/LX101/seat/`
2. `GET easyjet.com/flight/EZ222/seat/`
3. `POST swiss.com/booking`
4. `POST easyjet.com/booking`

the problem is not solved. Even if both step 1 and 2 may return a link to an available seat, the following booking requests may fail due to concurrent intermediate bookings. Thus, we may still end up in a situation where we have reserved one flight but not the other one. If 3 fails (due to, say, intermediate bookings at easyjet.com between steps 2 and 4) then we have one flight but not the other one. The retry of individual requests does not help here: we can try to repeat step 4 as many times as we like, but if the flight is fully booked then we will keep getting the same failure each time. What we really need is the ability to make step 3 and 4 tentative, so that they can be confirmed later. This way the whole process becomes atomic and happens entirely or not at all.

Our Goal: Lightweight Transactions for REST

The goal of this chapter is to propose a solution to the problem of atomicity within distributed RESTful interactions within the constraints of: (a) Using a lightweight transaction model (Pardon and Alonso 2000) based on ATOMIKOS TCC (Pardon 2009); (b) Minimizing, or in the best case, avoiding changes to the REST uniform interface and the HTTP protocol. (c) Assigning to the service running the composition the responsibility of ensuring the atomicity of the transaction.

A solution should provide the ability to transparently group multiple RESTful interactions and treat them as a single logical step, as well as to ensure that the consistency of a set of resources which are distributed over multiple servers can be kept. Whereas solutions have been proposed to batch interactions affecting multiple resources provided by a single server [e.g., WebDAV's explicit locking methods (Goland et al. 1999), or the transactions as a resource approach from (Richardson and Ruby, 2007, p. 231)], these are not directly applicable to interact with multiple resources distributed across multiple servers.

About this Chapter

This chapter contribution focuses on addressing the atomicity property of distributed transactions across RESTful Web services. This already satisfies the requirements of a wide class of applications, where atomicity is a necessity, while isolation is not. For example, all scenarios involving some kind of resource reservation are relevant, since once a resource is reserved within a transaction, its reserved state should become immediately visible to other clients in order to avoid overbooking. Our solution is thus applicable whenever clients need to atomically perform a purchase (or more in general, change the state) of a set of distributed and autonomous resources.

The rest of this chapter is structured as follows: in “A Transaction Model for REST” we use our running example to further define the business-driven case for REST transactions and then discuss the technical requirements that a solution should satisfy. In “Protocols” we outline the transaction protocol, which is discussed at length in “Discussion”. Finally, we give a brief survey of related work before drawing some conclusions.

A Transaction Model for REST

Whether or not REST needs transactions has been heavily debated within the REST community (Little 2009). We claim there is a clear need, and we try to motivate it here. Our motivation is in two parts. First, we define a business model for RESTful services that needs transactions. Next, we define the technical qualities that we think a transaction model for REST should possess in order to be successful.

Why REST Needs Transactions

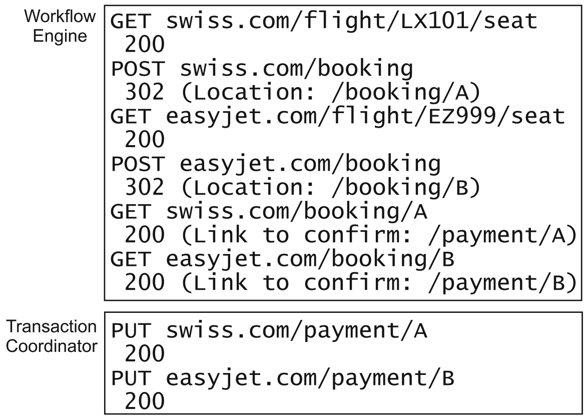
With the first story we have already motivated the need for atomicity, and why idempotent requests are not enough. We will now refine this model based on realistic business needs of each of the parties involved.

Refining our Example: Confirmation of Bookings

As hinted in the introduction, we need a way to make bookings tentative until confirmed:

Story 2 *As a customer, I want to be able to confirm a booking when I am done. Bookings that are not confirmed are not billed to my account.*

Confirmation can (and should) be business-specific. In the context of our running example, we assume that a confirmation hyperlink is returned by the RESTful API of the airline service (e.g., in response to a `GET/booking/{id}` the service returns `<flight number seat><payment uri="/payment/X"></flight>`). Thus, the booking can be confirmed with a `PUT/payment/X <VISA ...>` request.



```
GET swiss.com/flight/LX101/seat
 200
POST swiss.com/booking
 302 (Location: /booking/A)
GET easyjet.com/flight/EZ999/seat
 200
POST easyjet.com/booking
 302 (Location: /booking/B)
GET swiss.com/booking/A
 200 (Link to confirm: /payment/A)
GET easyjet.com/booking/B
 200 (Link to confirm: /payment/B)
```

```
PUT swiss.com/payment/A
 200
PUT easyjet.com/payment/B
 200
```

Fig. 23.1 Example of an atomic reservation for two flights (happy path)

Transactional Booking Workflow

The travel agency can now implement a transactional booking as shown by Fig. 23.1.

In terms of design, the first set of interactions can be driven by the workflow that composes the two services, while the final confirmations to close the transaction could be sent to a transaction coordinator component.

What if Step 4 Fails?

Let's return to the original problem: what if step 4 (i.e., the second booking) fails? By not performing any confirmation, the workflow engine ensures that no billing is done for either flight. This avoids our original problem as the transaction coordinator will not confirm any of the bookings.

Refining even more: Cancellation of Bookings

Confirmation is driven by the needs of the customer and the travel agency that composes the individual services. From the point of view of the airlines, an additional story arises:

Story 3 *As an airline, I do not want to wait for a confirmation forever. In other words, I want to be able to autonomously cancel a pending booking after some timeout expires.*

This should be obvious: as an airline, I do not want to loose money because some travel agency keeps seats reserved without confirmations. Consequently, we need a cancellation event triggered by some timeout specific to the airline.

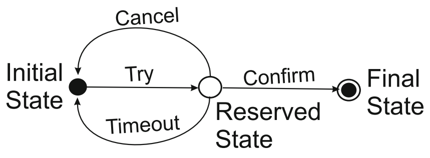


Fig. 23.2 Generic state machine of a resource complying with the Try-Cancel/Confirm protocol

The REST implementation could be as follows: `GET /booking/{id}` returns `<flight` number `seat><payment uri="/payment/X" deadline="24h"></flight>`). The composing workflow service can use the deadline as a hint to when the expiry of the reservation will happen.

Generalisation: Try-Cancel/Confirm

Our stories are particular illustrations of the more general Try-Cancel/Confirm (TCC) protocol. As shown in Fig. 23.2 each request is "tried" and remains tentative until it is either confirmed or cancelled. Composition of TCC services leads to a natural, loosely-coupled transaction model. Cancellation may occur spontaneously after a timeout or might be triggered by an external event (the latter we consider out-of-scope for this chapter).

Although originally formulated by Pardon (2009), a similar model seems to have been discovered independently at Amazon (Helland 2007) – which supports our vision about TCC's broad applicability and relevance.

Technical Requirements for REST Transactions

Industry practice has shown that transactions need to be non-invasive or they will be avoided. This is mostly due to the tight coupling and the additional complexity they introduce in the design and implementation of services which can participate in a transaction.

Our simple proposal attempts to avoid the negative impacts of existing approaches while ensuring that the visibility and the interoperability that have come to be expected of RESTful services are not affected.

Loose Coupling

The resources published by a RESTful Web service are typically seen as independent entities whose state changes happen autonomously from one another (Richardson and Ruby 2007). Clients interacting with resources may change their state trough atomic interactions which however do not span across multiple resources (Helland 2007).

The main constraint for our proposal is to ensure that resources remain autonomous and that performing transactions over them does not introduce any additional coupling among them. This is important to remain within the scope of the REST constraints which emphasize the role of the client as the one driving forward the state of an application.

A transaction solution for REST is considered loosely-coupled (Pautasso and Wilde 2009) if participating services are unaware of the fact that they are being part of a global transaction. More precisely: the individual participating services do not need to have any additional knowledge or implement any extra protocols besides what they already support. Whereas not all RESTful services may be able to participate in such transactions, we claim that there is a significant number of resources that naturally fit with our assumptions due to the nature of the business service they implement. This is particularly true for services that comply with the TCC business model outlined in the previous section.

No Context Please

Avoiding to make use of an explicit transaction context is a radical departure from most distributed transaction protocols which assume that a transactional context needs to be established and maintained among the participants, which must be aware of the transaction and thus become tightly coupled with one another.

One of the most important requirements to ensure loose coupling is that there should be no transaction context shipped around, thereby eliminating a lot of shared state interpretation and hidden dependencies among services. Most existing protocols for distributed transactions rely on such mechanism to establish a context shared among the participants. Thus the services become aware of participating in a transaction and must carry the burden of maintaining such context. Our goal is to define a protocol which removes the need for establishing and maintaining such context.

Align with the Business Functionality

The classical ACID transaction paradigm revolves around database locks and low-level rollback at the database level (Bernstein et al. 1987; Gray and Reuter 1993). Distributed ACID transactions (i.e., involving more than one database backend and/or service) usually require a "distributed transaction coordinator" to drive the individual ACID transactions via the XA protocol.

A lot has been said about the blocking nature of XA (Open Group 1992) and two-phase commit in classic ACID transaction technology – we will not repeat that here. Suffice it to say: any successful transaction technology for SOA should avoid the distributed locks associated with XA. The most natural way of doing this is with TCC (Pardon 2009). Instead of introducing long-running ACID transactions, this allows us to use multiple, short-lived ACID transactions for each of the resource state transitions triggered by the "try", "cancel" and "confirm" events (Fig. 23.2). In addition to avoiding lots of problems, service-specific confirm and cancel logic are also natural with respect to the business model of the service provider. This in turn means that transaction models embracing these will be less invasive and therefore more likely to be used.

Protocols

We will now introduce a set of protocols that ensure transactional correctness in REST systems. Let's start by defining the transaction a bit more formally:

Definition 1. A REST-based transaction T (e.g., booking a composed flight) is a number of invocations R_i (e.g., booking individual flights) across RESTful services S_i (e.g., swiss.com and easyjet.com) that need to either confirm altogether or cancel altogether. In other words: either all R_i succeed via an explicit confirmation $R_{i,confirm}$ (e.g., by paying for the flight), or all R_i cancel but nothing in between.

The Happy Path

1. A transactional workflow T goes about interacting with multiple distinct RESTful service APIs S_i
2. Interactions R_i may lead to a state transition of the participating service S_i identified by some URI – this URI corresponds to $R_{i,confirm}$
3. Once the workflow T successfully completes, the set of confirmation URIs and any required application-level payload is passed to a transaction service (or coordinator)
4. The transaction service then calls all of the $R_{i,confirm}$ with an idempotent `PUT` request on the corresponding URIs with the associated payloads

The protocol (Fig. 23.3) guarantees atomicity because each participating service receives a consistent request to either cancel or confirm. All participating services terminate their business transaction in the same way.

Note that we assume that $R_{i,confirm}$ is idempotent. In REST, this is a natural assumption to make. In practice, this means that the confirmation URI is called with a `PUT` or `DELETE` method – the particular choice depending on the contract defined by S_i and known to the workflow application, Fig. 23.4 illustrates this in the context of the running example.

Recovery Protocol

The basic protocol is very simple so it is natural to ask how this can work even in the presence of failures and recovery. Recovery is outlined below. We assume that each party is able to restore its own durable state, so we focus on the recoverability of the atomicity property across all parties.

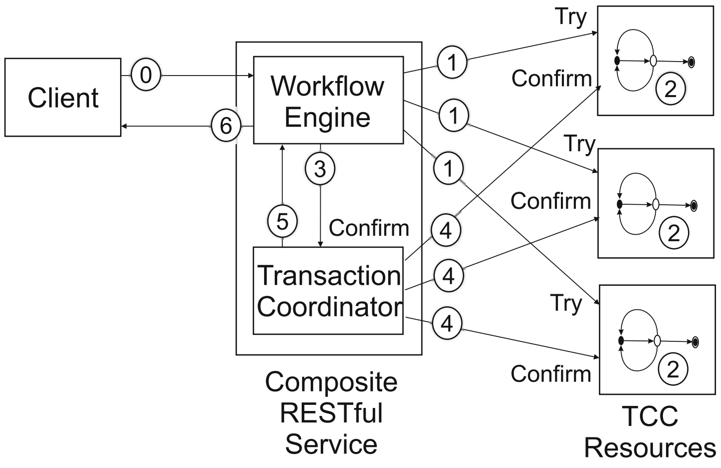
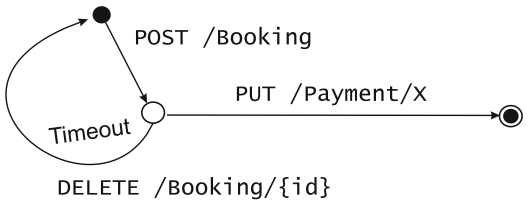


Fig. 23.3 Protocol architecture for the happy path

Fig. 23.4 Example of a flight reservation resource complying with the TCC pattern



Defining Recovery

For practical purposes, we define recovery as follows:

1. Checking the state of a transaction after node failure followed by restart, or
2. Checking the state of a transaction triggered by timeout

Recovery is something that is performed by the coordinator service as well as the participant services. For the coordinator this is expected since it intends to recover the transaction T that it knows about. For a participant, recovery also happens naturally: although a participant is not aware about T (following the loose coupling requirement), a participant service will want to release its reserved resources at the earliest possible time (as required by the business-level service contract).

Participant Service Recovery

Each participating service S_i does the following:

1. For recovery before step 2, do nothing.
2. For recovery after step 4: do nothing.
3. For recovery in between steps 2 and 4: execute $R_{i\ cancel}$ autonomously (This can be triggered by a timeout).

Coordinator Recovery

Like the participant service, we assume that the coordinator service is capable of restoring its durable state. Consequently, we focus on the recoverability of the overall atomicity. The coordinator has a slightly more complex job than the participants, because it has to make sure that all the participants will eventually arrive at the same end state for the transaction T. In particular, step 4 actually involves multiple participants so a failure during step 4 could be problematic[1]. We propose a naive protocol here, and leave optimisations to future work.

1. For recovery before step 2, do nothing.
2. For recovery between steps 2 and 4: do nothing.
3. For recovery after step 4: do nothing.
4. For recovery during step 4: retry $R_{i,confirm}$ with each participating service S_i. Since $R_{i,confirm}$ are performed using idempotent methods, they may be retried as many times as necessary. Note: this requires the coordinator to durably log all participant information before starting step 4.

Discussion

This section presents reflections on the proposed protocols. In particular, we show that they can guarantee atomicity even in the event of failures and outline the known limitations of the approach.

Atomicity Guaranteed even with Failures

Even if there are arbitrary failures, we still preserve atomicity – eventually. In other words: given enough time, the global transaction T will be confirmed everywhere, or cancelled everywhere, or nothing will have happend in the first place. More precisely: either all R_i are confirmed, or all are cancelled. In order to prove this, we take a closer look at the protocol steps from the point of view of the coordinator. Here is our proof:

1. If there are no failures, then steps 1–4 run through and each R_i will have been confirmed.
2. For any failures before step 2, no R_i exists, meaning that nothing has happened.
3. For any failures during or after step 2 but before step 4: all R_i will eventually be cancelled autonomously by each S_i (since nothing has been confirmed yet).

[1]Especially because the participants are not aware that they are part of a transaction

4. For any failures during step 4: the coordinator will retry each $R_{i,confirm}$ until it succeeds. Because confirmation is idempotent, this will eventually succeed (note: there is one caveat here – discussed next).
5. For any failures after step 4: all $R_{i,confirm}$ have been done, so we already have atomicity and no action is required.

The Exception that Confirms the Rule: Heuristics

There is one weak spot in our proof of atomicity: during step 4, some service S_i may time out and cancel on its own, while the coordinator is performing confirmation. In the worst case, this means that some participants confirm whereas others cancel on their own – effectively breaking atomicity. We call this a *heuristic exception* for reasons outlined in the following.

Perfection does not Exist

There has been a lot of interesting work related to atomicity, and the more general problem of distributed agreement, and the most important result is that a perfect solution is not possible (Fischer 1985). In practice, this means that there is always the possibility that at least one participating service/node is unaware of the outcome of the "global" distributed transaction - be it with our TCC protocol for RESTful Web services or with classic, ACID, XA-style transactions.

The practical consequence is that one or more nodes can remain "in-doubt" about the global result of one or more business transactions that they are participating in. For instance, flight reservations may never complete because payment never arrives (either due network failures, node failures or both).

This is not specific to REST or WS-*, it exists in any networked environment: there is no perfect protocol for distributed agreement. This is a limitation one has to live with (and one of the drivers behind the CAP theorem Brewer 2000).

Enter Heuristics

The bottom line is that perfect atomicity may not be possible sometimes, and we need a practical way of dealing with such scenarios (just like workflow-based solutions do). We propose a simple way based on the "heuristic exceptions" known from the industry's two-phase commit protocol families (such as OTS Ram et al. 1999).

In practice, most industrial distributed two-phase commit (2PC) technologies recognize that similar anomalies may happen. In order to avoid that a participant remains in-doubt about the outcome, these protocols allow the participants to timeout and unilaterally terminate their part of the a global transaction with a so-called “heurisitic decision” (e.g., heuristic rollback).

Our Protocol Compared to Two-Phase Commit

Once a participant completes R_i, it can be considered in-doubt: all its durable state changes are on disk, and the only remaining thing is the pending confirmation $R_{i,confirm}$ on behalf of T. If the participant decides to time-out then this is similar to what classical two-phase commit calls a heuristic rollback. The default way of handling this is very similar: we make sure that the coordinator logs this fact on behalf of T and assume that this will be reported in some implementation-specific way to allow for out of band manual resolution of the inconsistency by a human operator.

Advantages of our Protocol Compared to Classical Two-Phase Commit

One big advantage our protocol offers (compared to classical heuristic cases) is the fact that it offers higher-level semantics and does not hold low-level database locks. In-doubt participants do not block any other work other than the one affected by the business resources they reserve on behalf of T. When a heuristic cancel is done by S_i, the consequences are well-defined and known to the business: it corresponds to a unilateral breach (by S_i) of the contract entered into with the execution of R_i. Both the coordinator of T and the site administrators at S_i can use the high-level information to manually resolve the inconsistency. Contrast this to classical transactions, where heuristic exceptions are very low-level error conditions with vague impact and little context information. In this way, our protocol embraces the fact that distributed agreement between businesses is challenging due to the inherent limitations of distributed agreement and the CAP theorem.

Optimisations and Future Work

We have presented a simple protocol that ensures atomicity in at least as many cases as ACID transactions do, without the restrictions. However, there is a lot of room for optimisation. We can see at least the following things to refine:

1. Optimising the basic protocol with coordinator-driven cancellation in addition to confirmation. This allows the application/workflow to signal failures early, so that participating services do not have to time out. This in turn minimises resource contention.

2. Optimising timeout management by the coordinator in order to minimize the occurrence of heuristic exception cases. For instance, the coordinator could inquire (GET) with each participant to discover the remaining timeout before attempting to confirm. If the remaining timeout is below a threshold, then the coordinator might refuse to even start confirmation.
3. Optimising the handling of heuristic exceptions if they do happen. For instance, the coordinator could inquire at each participating service to find out more about what to do, or a management-by-exception type of workflow could be triggered that requires human intervention at the workflow end. This sounds all the more interesting because it is backed by the way that real businesses work today.
4. Our basic assumptions could be weakened. For instance, it might be that some service providers do not hold reservations. Likewise, it might be that some requests cannot fail under normal circumstances (like read-only `GET` requests). Further research along these lines, will help to widen the applicability of transactions over RESTful APIs which do not fully comply with the Try-Cancel/Confirm pattern.

Related Work

RESTful Service Composition

REST is widely perceived as an emerging lightweight technology for building Web services (Richardson and Ruby 2007). The properties of the REST architectural style are meant to enable *serendipitous reuse by means of composition* (Vinoski 2008).

The idea of RESTful service composition has also been explored in the Bite project (Rosenberg et al. 2008), or with the BPEL for REST extensions (Pautasso 2008). Also, Xu et al. (2008); Pautasso (2009) proposes to use workflow languages for composing RESTful services. All of these contributions to do not explicitly address the requirement for transactional composition of RESTful services.

RESTful Transaction Models

In addition to several threads on the *rest-discuss* mailing list, summarized by Little (2009), the problem of transactional interactions for RESTful services has started to attract some interest also in the research community. For example, an approach to RESTful transactions based on isolation theorems has been recently proposed in Razavi et al. (2009). The RETRO (Marinos et al. 2009) transaction model also complies with the REST architectural style.

REST-*

The recently appeared book "REST in Practice" (Webber et al. 2010) also has a dedicated chapter discussing transaction support for REST. The approach seems similar to what REST-* is trying to accomplish, with the same drawback of tight coupling due to, among other things, a transaction context going all around.

More in detail, the JBoss REST-* initiative aims at providing various QoS guarantees for RESTful Web services, in much the same way as WS-* has done for Web services by creating a "stack" of agreed upon best practices and standards for REST middleware. In its attempt at offering transactions, REST-* follows an approach that is reasonably close to TIP and WS-AT: a context is added to each invocation in order to make the invocation transactional. The receiving service has to understand that context in order to participate in the transaction outcome. This leads to tight coupling, something that we have tried to avoid.

ATOM Pub/Sub

Another common approach for reaching distributed agreement in REST uses a publish/subscribe mechanism based on feeds wherein the "transaction coordinator" publishes updates on the "outcome" of the transaction, and each participant then listens for any updates it might be interested in. This is certainly technically feasible, however it assumes that each participant knows the right feed that should be subscribed to, and understands the semantics of the updates being published by the coordinator. In our solution, the participants do not have to know anything besides their own business contract (API). Thus, we believe our approach introduces less coupling than this one.

Also, a publish/subscribe mechanism implies that the coordinator has no direct means of asking a participant service about its final outcome (taking into account any heuristic decisions it may have taken after a timeout). This seems a bit awkward to us.

Distributed Transaction Technologies

This section provides some relevant background information on related transaction technologies/standards for Internet-scale systems and/or service-oriented architectures.

TIP

The TIP (Transaction Internet Protocol) was one of the first initiatives to offer reliability on the wider scale of the Internet (Vogler et al. 1999), and across different service providers. It is based on the notion of a transaction context that is passed along with each request. The notion of such a context is far from ideal because it introduces tight coupling and limits the interoperability of the participants.

CORBA OTS

Within the CORBA ecosystem (Henning 2006), the OTS (Object Transaction Service) is a distributed transaction framework that (at least in theory) provides interoperability of transactions across CORBA objects and even across ORBs (Ram et al. 1999). It is used primarily in financial and telecom industries and it allows for a certain heterogeneity. However, as every system based on binary ORB protocols and bindings, CORBA/OTS cannot be directly reused in the domain of RESTful Web services.

WS-*

The WS-* stack would not have earned its fame if it did not offer some form of transaction support. A number of competing standards have been proposed (Zimmermann et al. 2007), but all of them were designed by committee. This implies that they all tend to be somewhat over-engineered, and above all they are driven by technology vendors (Tai et al. 2004) rather than by practical needs or demands. Consequently, their practical relevance is rather limited.

The two most common approaches are the following: WS-AT and WS-BT. We will discuss them starting from the assumption that the main value proposition of the WS-* technology stack lies in its intrinsic interoperability between heterogeneous platforms.

Web Service – Atomic Transactions (WS–AT) is the WS-* counterpart of the classical ACID transaction technologies. It offers distributed XA transactions over web service protocols.

As far as we know, this is the only transaction standard that enjoys real cross-vendor support from the bigger players like IBM and Microsoft. Unfortunately, this complex specification leads to tight coupling between participating sites. Configuration is not easy, especially if security is involved. Interoperability among existing implementations has also been difficult to achieve.

Web Service – Business Activity (WS–BA) is a compensation-based protocol that arose out of the BPEL world as a way to make BPEL engines coordinate compensation scopes across vendors/engines. It offers the possibility to "compensate" for unrightfully executed work with application-level callbacks. However, there is no notion of a business-level "confirmation" phase, which may be needed to address our requirements.

We do not know of many vendors who support this standard. Microsoft, for instance, does not. This makes the usefulness for interoperability rather limited and hence the relevance of this technology may be questionable.

XA Technology

The XA (Open Group 1992) specification defines an open, vendor-independent way of supporting distributed ACID transactions across back-end systems. It is the classical way of doing distributed transactions a distributed system – but due to tight coupling limitations it is too restrictive for service-oriented architectures and REST.

Try Confirm/Cancel

Try Confirm/Cancel (TCC) is a business-level protocol for distributed atomic transactions offered by Pardon (2009). The main difference with the previously described approaches is that the transactional events corresponding to cancel ("rollback") and confirm ("commit") are not defined by the needs of the middleware/database but rather by the application/business services[2]. This makes TCC a highly practical and business-oriented protocol, which – as we have shown in this Chapter – fits very well within the constraints of the REST uniform interface.

Although the current implementations by Atomikos are based on protocols such as RMI/IIOP and WS-*, the underlying ideas lend themselves very well to RESTful Web services, without the need to introduce coupling. In fact, applying TCC to REST allows to offer distributed transactions with services that are unaware of being part of such atomic transaction.

Conclusion

In this chapter, we propose a light-weight atomic transaction solution for REST based on applying the Try-Cancel/Confirm (TCC) pattern to the design of a RESTful Web service. The pattern fits with the business requirements of many service providers (e.g., e-Commerce sites) that need to participate within long running transactions and thus offer services allowing clients to issue requests which can later be canceled and have to be confirmed within a given timeout before they are carried out.

In addition to defining the business case for REST transactions, we have proposed a simple protocol to achieve atomicity among distributed resources that comply with the TCC pattern. We illustrated the protocol's behaviour with an example also showing that the resources involved in the transaction remain unaware of the transaction. Finally we have discussed how the protocol provides a loosely coupled solution to guarantee atomicity and consistency in the event of failures and outlined the known limitations (shared by all distributed agreement protocols) mainly due to heuristic timeouts.

[2]A similar idea (but lacking the "try" phase) was also proposed in the OASIS BTP proposal (Dalal et al. 2003), which was standardized but remains without any current implementations.

References

Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

Eric A. Brewer. Towards robust distributed systems (abstract). In *Proc. of the 19th Annual ACM Symposium on Principles of Distributed Computing*, page 7, Portland, Oregon, July 2000.

Sanjay Dalal, Sazi Temel, Mark Little, Mark Potts, and Jim Webber. Coordinating Business Transactions on the Web. *IEEE Internet Computing*, 7(1):30–39, January 2003.

Roy Fielding. *Architectural Styles and The Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

Yaron Y. Goland, E. James Whitehead, A. Faizi, S. Carter, and D. Jensen. HTTP Extensions for Distributed Authoring — WebDAV. Internet RFC 2518, February 1999.

Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

Pat Helland. Life beyond Distributed Transactions: an Apostate's Opinion. In *Third Biennial Conference on Innovative Data Systems Research (CIDR 2007)*, pages 132–141, Asilomar, CA, January 2007.

Michi Henning. The Rise and Fall of CORBA. *ACM Queue*, 4(5):28–34, June 2006.

Mark Little. *REST and transactions?*, 2009. http://www.infoq.com/news/2009/06/rest-ts.

Alexandros Marinos, Amir R. Razavi, Sotiris Moschoyiannis, and Paul J. Krause. RETRO: A Consistent and Recoverable RESTful Transaction Model. In *Proc. of the IEEE International Conference on Web Services (ICWS 2009)*, pages 181–188, Los Angeles, CA, USA, July 2009.

Open Group. Distributed TP: The XA Specification, February 1992.

Guy Pardon. *Try-Cancel/Confirm: Transactions for (Web) Services*, 2009. http://www.atomikos.com/Publications/TryCancelConfirm.

Guy Pardon and Gustavo Alonso. CheeTah: a Lightweight Transaction Server for Plug-and-Play Internet Data Management. In *Proceedings of 26th International Conference on Very Large Data Bases (VLDB 2000)*, pages 210–219, Cairo, Egypt, September 2000.

Cesare Pautasso. BPEL for REST. In *7th International Conference on Business Process Management (BPM08)*, Milan, Italy, September 2008.

Cesare Pautasso. Composing RESTful Services with JOpera. In *Proc. of the International Conference on Software Composition (SC09)*, pages 142–159, Zurich, Switzerland, July 2009.

Cesare Pautasso and Erik Wilde. Why is the Web Loosely Coupled? A Multi-Faceted Metric for Service Design. In *Proc. of the 18th International World Wide Web Conference*, pages 911–920, Madrid, Spain, May 2009.

Prabhu Ram, Lyman Do, Pamela Drew, and Tong Zhou. Object Transaction Service: Experiences and Open Issues. In *International Symposium on Distributed Objects and Applications (DOA 1999)*, pages 296–304, Edinburgh, UK, September 1999.

Amir R. Razavi, Alexandros Marinos, Sotiris Moschoyiannis, and Paul J. Krause. RESTful Transactions Supported by the Isolation Theorems. In *ICWE'09*, pages 394–409, 2009.

Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly, May 2007.

Florian Rosenberg, Francisco Curbera, Matthew J. Duftler, and Rania Kahalf. Composing RESTful Services and Collaborative Workflows. *IEEE Internet Computing*, 12(5):24–31, September-October 2008.

Stefan Tai, Thomas Mikalsen, Eric Wohlstadter, Nirmit Desai, and Isabelle Rouvellou. Transaction policies for service-oriented computing. *Data Knowl. Eng.*, 51(1):59–79, 2004.

Steve Vinoski. Serendipitous Reuse. *IEEE Internet Computing*, 12(1):84–87, 2008.

Hartmut Vogler, Marie-Luise Moschgath, Thomas Kunkelmann, and J. Grünewald. The Transaction Internet Protocol in Practice: Reliability for WWW Applications. IEEE Computer Society, Internet Workshop'99 (IWS'99), February 1999.

Jim Webber, Savas Parastatidis, and Ian Robinson. *REST in practice*. O'Reilly, September 2010.

Xiwei Xu, Liming Zhu, Yan Liu, and Mark Staples. Resource-Oriented Architecture for Business Processes. In *Proc of the 15th Asia-Pacific Software Engineering Conference (APSEC2008)*, December 2008.

Olaf Zimmermann, Jonas Grundler, Stefan Tai, and Frank Leymann. Architectural Decisions and Patterns for Transactional Worlflows in SOA. In *Proc. of the 5th International Conference on Service-Oriented Computing*, Vienna, Austria, 2007.

Index

DOI 10.1007/978-1-4419-8303-9, © Springer Science+Business Media, LLC 2011