Java EE Enterprise Application

Case Study

Complete Solution addressing NFRs

Java EE Enterprise Application
Case Study
Complete Solution addressing NFRs

ISBN: 9781483544731

Preface

The motivation to write this book comes from facts that there a gamut of frameworks and technologies today to build a custom enterprise applications using Java frameworks. Which options should I go with and what are the choices that I need to make. Do I need to procure multiple frameworks or Java EE (5 and above) provides comprehensive capabilities to build an enterprise application. Understanding this and the pros and cons of the several J2EE technologies how they fit into the Java enterprise application is the subject of this book. This book also addresses the NFRs and explains how they will be addressed in the solution. This title provides a step by step approach and best practices to help architects and SME evangelize the solution which cover various views/diagrams including design decision, assumptions, and risk and mitigation actions.

There are no reference artifacts in today that covers the entire process from requirements to solutioning for an enterprise grade Java Application. These references focus on describing the frameworks and its capability but none provides an end-2-end view and process of how the solution can be evangelized including addressing NFRs. This critical areas covered are transaction, security, frameworks, hardware sizing, deployment models and scaling of the applications. Most of the books and references will provide theoretical approach giving only limited guidance which is not adequate for real world scenarios or enterprise scale applications. The book is a good reference for understanding how the UML notations and various models are leveraged by SME, designers and architects to document enterprise applications.

This book describes the process of building a full scale real world solution using the JEE (5.0 and above) application framework.

Acknowledgements

Thank you to my discussion partners, reviewers and supporters, whose valuable comments and feedback have greatly contributed to this book. I look forward to your feedback on this title on an on-going basis.

Special thanks to my wife Vaishali and family members for the encouragement and constant support during the development of the book.

About the Author

Sameer S Paradkar is an Enterprise Architect with has 15+ years of extensive experience in the ICT industry which spans across Consulting, Product Development and Systems Integration. He is certified in Open Group TOGAF, Oracle Master Java EA (SCEA), TMForum NGOSS, IBM SOA Solutions, IBM Cloud Solutions and ITIL Foundation V3. He serves as an advisory architect and mentor on Enterprise Architecture initiatives and continues to work as a Subject Matter Expert and author. He has worked on multiple architecture transformation engagements in the USA, UK, Europe, Asia Pacific and Middle East Regions that presented a phased roadmap to transformation that maximized the business value, while minimizing cost and risks. He has been a globe-trotter and has travelled to multiple countries in 5 continents in his ICT career.

• He has worked as an enterprise architect on several IT architecture transformation and modernization projects worldwide. Past clients includes AVIVA UK, AOL US, BT UK, Citigroup - APAC, LIC India, Lloyds Bank UK, Microsoft US, Mobily Saudi Arabia, Saudi Ports Saudi Arabia, Siemens AG Germany, STC Saudi Arabia, Telecom New Zealand, Telenor Asia Pacific, Telstra Australia, Tunsiana Telecom, Walgreens US.

• He has engaged with Department Heads, Technical and Business leader's levels in the past and advised them on

technology adoption strategy and roadmaps. He has worked as Enterprise Architect as part of the Enterprise Architecture Consulting advisory practice focusing on assessments, strategy alignment, business case development and governance across industry verticals.

• He has written several articles over the last few years that have been published worldwide including BPTrends, BPMInstitite and SOAInstitute.

• He has also worked as Consultant with focus on EAI, SOA, and BPM envisaging and building IT systems and business-critical applications for Fortune 100 customers - for major Retail, Banking, Telecom, Insurance, and Healthcare verticals.

• He has proven successful experience leading and driving multiple large scale architecture modernization programs for several years.

Prior to **EY - IT Transformation - IT Advisory** He has worked in organizations like **IBM GBS, Wipro Consulting Services**, **Infosys Technologies** and **Deloitte Consulting** and specializes in IT Strategies and Enterprise transformation initiatives.

Contents

List of Figures

Part I - Solution

Introduction

This book presents a specific scenario similar in complexity to that you can expect in a real world for enterprise customers, and then presents a step by step approach to creating the Java Application. The solution is built using Java EE frameworks and technologies. Initial context around the business problem is presented which includes business requirements, use cases and NFR expected for the solution. As an architect you will be required to apply your analytical abilities and knowledge of Java technologies and frameworks to come up with the best solution for the given problem statement while making the right design choices where ever required. As an architect you will be required to make assumptions, design decision and identify risk and mitigation actions which are also captured and covered in this artifact.

Non-functional requirements are critical to the success of project and NFRs are the life-line of any software application. Capturing and addressing them is a critical activity in any IT project. This book provides comprehensive details for analysis and solutioning of non-functional requirements for the enterprise Java Application.

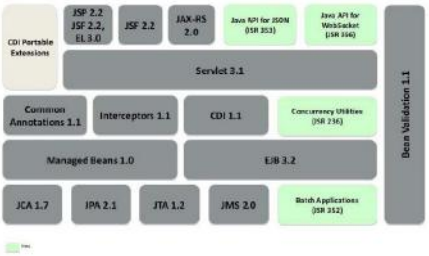

Figure 1: Java EE Framework Components

Tackling Business Problem

The analyst's team will come up with a brief introduction to the customer background and the needs for a Java EE application. They will include a structured description of the main requirements, a domain model, use case diagrams and NFRs and you are expected to deliver the solution for the Java application using the Java EE technologies and framework.

The business problem captured or discussed during workshops might be vague and may specify very high-level description of the main use cases and thus there will be many open questions or areas where further deep dive should be planned. The requirements may be incomplete namely the use case diagrams, and miss information necessary for the design. Therefore you have to fill the missing parts for yourself and replace the missing information with assumptions, perhaps adding the unclear element to the list of risks. This is best done through workshops, interview and discussion with various stakeholders from business and technology departments. Once the proper levels of details are available and you have the solution in your mind, it isn't too difficult to draw the views for the solution using a suitable Visual editor. One can leverage tools like MS Visio for the diagrams as they are very intuitive and more than solves the purpose.

The key part of the solution itself the various diagrams or the views are the key outputs delivered by the architects. As an architect you have to decide which components or interfaces the system should have their integration into the eco-system

and how the non-functional and functional requirements will be addressed.

The process will also involve a considerable amount of time estimating the resource (Hardware) requirements of the application and exploring and comparing deployment environments and the non-functional characteristics for various physical vs the cloud options. This section will cover this methodology and approach that you should apply while creating your Java solution.

The solution typically includes among other things

• Class Diagram

• Component Diagram that describes various tiers and components in scope

• Deployment diagram that describes the proposed physical layout of the major tiers

• Sequence or Collaboration diagram for each use case

• Technical risks & mitigation strategies

• Design Decision that are made for the Solutions

• Assumptions made for the Solution

Risks and Assumptions

In this segment you are tested on the ability to recognize the top technical risks present in the given business scenario and

the technology landscape and how these will be addressed by your solution. Be strategic and objective in your assessment of risk. Low-level risks should be avoided and high-level, systemic risks that would be nightmare scenarios should be included in this section. E.g In an online auctioning application, a security limitation in any form would be a major risk than worrying about if the application could run in multiple browsers. The mitigation and hopefully complete removal will be the focus of this section.

Class Diagram – from Domain Model to Class Diagrams

One has to ensure that all the objects detailed in the domain model are present in the class diagram. Select and commit to a method or framework that describes how you plan to build the solution at the web/presentation, business logic, persistence, and integration tiers. There are some important points to note on the class diagram, as follows:

• The level of details in the class diagram will include the entities, important manager EJBs and few other classes representing important architectural concepts (a cache, an interceptor, a DAO for accessing an external tool etc. which needs to be addressed by the solution)

• The domain model will map to classes nearly directly, only on a few occasions modifications and extensions will be required e.g. introducing a base class.

• Annotations will be used to show how specific items in the domain model are mapped onto JEE components specifically, session beans and Entity classes. The class diagram will

contain important information pertaining @Entity or @Embeddable, where ever applicable.

• The class diagram remains web framework agnostic. Any web framework can be leveraged to build the solution.

Class diagram may also contain types, packages, relationships, and even instances such as objects and links. Class diagrams are static; they display what interacts but not what happens when they do interact. A class diagram can have three kinds of relationships:

• Association is a relationship between instances of the two classes. An association exists between two classes if an instance of one class must know about the other to perform its work. In a diagram, an association is a link connecting two classes.

• Aggregation is an association in which one class belongs to a collection. An aggregation shows a diamond end pointing to the part containing the whole.

• Generalization is an inheritance link indicating one class is a superclass of another. A generalization shows a triangle pointing to the superclass.

Using the initial list of business classes, you develop class diagrams by identifying and defining the relationships among the classes from the domain model. This is done in an interactive development workshop with business partners. It is also useful to keep these diagrams on display on a whiteboard or other medium, and to develop it gradually as the project progresses. The diagrams are stored on a UML

tool to provide access to all team members and other interested parties. The class diagrams are also shows relationships among classes. This aspect of the diagrams may tend to evolve later in the design process, as lower level classes are identified. The class diagrams will improve the definition of the classes, which in turn may require changes to the sequence diagrams and, when developed, the state transition diagrams. These other diagrams will also have an impact on the class diagrams.

Component Diagram – Logical Architecture

The component diagram is another view of the system, at a higher level than the class diagram. In this view, you are expected to demonstrate the ability to visualize the system at a higher level and understand all of the critical parts in your solution. If you have proposed a MDBs to solve a particularly integration issue, here is where you need to depict in the diagram. The component diagram represents the high-level parts that make up the modeled application. This diagram is a high-level depiction of the components and the relationships. A component diagram depicts the components' refined post-development or construction phase.

Component diagrams are physical versions of class diagrams. A component diagram shows the relationships and dependencies between software components, including Java source code components, Java class components, and Java deployable components JAR (Java Archive) files. Within the deployment diagram, a software component may be represented as a component type. With respect to Java EE some components exist at compile time, some exist at archive time, some exist at runtime and some exist at more than one

time. A compile-only component is the one that is meaningful only at compile time similarly the runtime component in this case would be an executable program.

Deployment Diagram – Physical Architecture

The deployment diagram captures information about the infrastructure characteristics. Naming specific machines, vendors, or routers may not be essential, as these decisions change so quickly. Do indicate a vendor/machine-agnostic way the resources you expect to be deployed in order to support the architecture namely CPUs, RAM, network requirements, disk configuration, and so on and then provide concrete examples of a specific vendor/machine combination that satisfies your theoretical capacity prediction. Although the resources deemed necessary will vary from one scenario to the next, the fundamental resources themselves will not. These are as follows:

• CPUs (number of cores, clock speed)

• RAM (quantity in GB)

• Network (minimum interface speed)

• Storage (disk/SAN configuration)

Deployment diagram puts it altogether and captures the configuration of the runtime elements of the application. This diagram is obviously most useful when an application is complete and ready to be deployed. Depicts the nodes i.e., a JEE server, a database server it accesses, and the user workstation used to access the JEE application.

Deployment diagrams show the physical configurations of software and hardware. The deployment diagram complements the component diagram. It shows the configuration of runtime processing elements such as servers and other hardware and the software components, processes, and objects that they comprise. Software component instances represent runtime manifestations of classes. Components that do not runtime entities do not appear on these diagrams they are shown on component diagrams. Deployment diagram is a graphical representation of nodes connected by communication links or associations. Nodes may contain component instances, which indicate that the component resides and runs on the node. The deployment diagram can be used to show which components run on which nodes. The migration of components from node to node or objects from component to component may also be represented.

Sequence Diagram – from Use Cases to Sequence Diagrams

Sequence diagram describes how groups of objects collaborate in some behavior over time. It records the behavior of a single use case. It displays objects and the messages passed among these objects in the use case. A design can have lots of methods in different classes. This makes it difficult to determine the overall sequence of behavior. This diagram is simple and logical, so as to make the sequence and flow of control obvious.

The sequence diagram shows the explicit series of interactions as they flow through the system to address the desired objective or result. The sequence diagram is especially useful in systems with time-dependent

functionality (such as real-time applications) and for complex scenarios where time dependencies are critical. It has two aspects namely Time and Various objects participating in a sequence of events required for a purpose.

Usually the sequence of events to which the objects of the system are subject is important in real-time applications, the time axis is an important measurement. This view identifies the roles of the objects in your system through the sequence of states they traverse to accomplish the goal. This view is an event driven perspective of the system. The relationships among the roles are not shown. Class and object diagrams present such static views. Sequence and interaction diagrams are dynamic. They describe how objects collaborate or interact. A sequence diagram is an interaction diagram that details the functionality and messages (requests and responses) and the timing. The time progresses as you move down the page. The objects involved in the operation are listed from left to right according to when they take part in the message sequence. The sequence diagram can have a clarifying note, text inside a dog-eared rectangle. Notes can be put into any kind of UML diagram.

Sequence diagrams are models of business processes that represent the different interactions between actors and objects in the system. Each process has a process owner and goals (such as cycle time, defect rate, and cost) and consists of a set of business activities (in sequence and/or in parallel).

Sequence or collaboration diagram should be covered for each specified use case. Do not roll one or more use cases together into a single diagram to save time. Sequence diagrams should be clear and broadly map to the complexity

of the use case. The classes and components you created in the class and component diagram should be represented, along with the calls between them necessary to implement the use case being documented.

Design Decisions

This section provides highlights on the various design decisions an architect has to make while building the enterprise Java solutions. This section covers all the critical areas including, transaction, security, frameworks, hardware sizing, deployment models and scaling of the applications.

Java EE Frameworks for Rescue

The common practice in J2EE 1.3 and J2EE 1.4 was indeed to supplement the J2EE stack with quite a lot of additional frameworks.

Starting from Java EE 5 and now certainly with Java EE 6 & 7 you don't need additional frameworks as JEE provides reference implementations for all the specs. Everything core Spring container provided to J2EE can now be done with the lightweight EJB3.1 and CDI component models, ORM support is provided by JPA which is often implemented by Hibernate. The MVC web framework is part of the standard stack via JSF.

Of course individuals may still prefer the Spring and Struts programming APIs, but with Java EE 6 there is no pressing need to use them. Or put differently, it's no longer absolutely needed to complement Java EE unless if there aren’t

corresponding reference implementation available from Oracle.

RedHat Seam does provide some nice (portable) extensions to Java EE and JSF was actually built to be used with extension and third party component libraries. Such a library is however not really a 'framework', but just gives you additional widgets to use on your pages and not the same as replacing JSF with say Struts or GWT.

	2014 (To-Be)	2006	2001
EE Specs	• Java EE 6, 7	• J2EE 1.4	• J2EE 1.2
SE Specs	• JDK 7, 8	• J2SDK 1.4	• J2SDK 1.3
Web	• JSF 2.0, 2.2 (inc. Facelet)	• Struts 1.3 / 2 + JSP 2.0 + JSTL 1.1	• Struts 1.1 + JSP 1.1 + commons taglib
Rich Client	• JavaFX 8	• SWT	• Swing (JFC)
Business Logic	• EJB 3.1, 3.2	• Spring FW • POJO • Struts Action	• EJB 1.1 • POJO • Struts Action
Data Access	• JPA 2.0, 2.1	• Hibernate • iBATIS	• Original JDBC wrapper
Logging	• Apache Log4J • SLF4J + Logback	• Apache Log4J • commons logging	• Jakarta Log4J

Figure 2: Java EE Framework Comparison

N-Tier or No-Tier

N-Tier Systems if well architected and designed can help achieve all non-functional service levels requirements of the system. Due to a highly distributed model, manageability may suffer a little in N-Tier systems, but since all systems are highly modular, the issue is fairly easy to address. It is also notable that Architects have to sometimes compromise a little on one service level requirement, to achieve the desired effect on another. For example Performance and Security show an inverse proportional relationship.

Consists of:

• Client Tier: The tier with which the end user interacts. Clients can be 'thin clients' as in the case of Browser based applications or fat clients as in the case of client Java applications.

• Web Tier: Decouples the client tier from the Business tier. Java Servlets and JSPs reside in this tier. Servlets act as Controllers; they translate incoming requests, and dispatch them to components that can invoke the business events in the Business Tier. JSP combine static templates with dynamic data to create dynamic output that the client tier uses for presentation to the user.

• Business / Application Logic Tier: Generally implemented using Enterprise Java Beans (EJB) that act as business process objects and business domain objects. EJB containers provide various services such as Object Distribution, Persistence, Transaction, Resource Management, and Security and so on.

• Enterprise Information System Integration Tier: EIS Integration tier interfaces between the Business (and sometimes Web tier) objects and Enterprise Information Systems. Example, Data Access Objects (DAO) decouples Enterprise beans (typically Session Beans or BMP Entity Beans) with Enterprise Data.

• Enterprise Information System Tier: This represents all the Enterprise data. This could be in many forms including Relational Databases, XML Databases, and ERP Systems and so on.

EJB Container Model

The EJB Container sits between an EJB Server and EJBs. An EJB Server can have one or more EJB Containers and each container can manage one or more components. The EJB Container manages the *EJBHome* and *EJBObject* implementations. Via these objects, the container decorates Enterprise Bean classes and provides various services such as:

• Life cycle management

• Naming

• Object Distribution

• Persistence

• Security

• Transactions

• Concurrency

The EJB Container also does resource management through Instance Pooling and swapping (in the case of Stateless Session Beans) and Passivation / Activation, in the case of Stateful Session Beans and Entity Beans.

Enterprise Java Bean Vs No Beans

To accommodate a growing number of users, you may need to distribute an application's components across multiple machines. The enterprise beans of an application can not only run on different machines, but also the location will remain transparent to the clients. Transactions are required to ensure

data integrity. Enterprise beans support transactions, the mechanisms that manage the concurrent access of shared objects. The application will have a variety of clients. With just a few lines of code, remote clients can locate enterprise beans. These clients can be thin, various, and numerous. Scalable applications can be building using EJBs.

EJB 3.0 has been split into two parts: the EJB spec and the JPA spec. EJBs are now known as entities in JEE 5 and later and are POJOs. JPA (Java Persistence API) is as the name suggests the persistence part of the API. This is commonly used in libraries like EclipseLink, Hibernate and Toplink. Hibernate is the most popular and predates JPA but the differences aren't so large. JPA is an ORM (object-relational mapper) for projecting an object model onto a relational database.

EJB 3 Improvements

• Simplified process of developing ejb reduces significantly overhead.

• Using Java Language annotations as configuration improves developer’s productivity.

• Specification of programmatic defaults, including for metadata, reducing the need for the developer to specify common, expected behaviors and requirements on the EJB container.

• Encapsulation of environmental dependencies and JNDI access through the use of annotations, dependency injection mechanisms, and simple lookup mechanisms.

• Simplification of the enterprise bean types.

• EJB support inheritance and polymorphism.

• Light-weight CRUD operations with JPA EnityManager API.

• Enhanced query JPA capabilities.

• Life cycle callback methods can be defined in ejb itself or in a bean listener class.

• Interceptor facility listeners for session beans and message-driven beans. An interceptor method may be defined on the bean class or on an interceptor class associated with the bean.

• It is possible to use both CMP and JPA in one application

Bean Managed Vs Container Managed Transaction

The Java Platform EE platform supports two transaction-management paradigms: declarative transaction demarcation and programmatic transaction demarcation. Declarative transaction management refers to a non-programmatic demarcation of transaction boundaries, achieved by specifying within the deployment descriptor the transaction attributes for the various methods of the container-managed EJB component. This is a flexible approach that facilitates changes in the application's transactional characteristics without modifying any code. Container-managed transaction demarcation must be used by entity EJB components.

In bean-managed transaction demarcation, the EJB bean uses *UserTransaction*. Only session beans can choose to use bean-managed transactions. In container-managed transaction demarcation, the EJB container is responsible for transaction demarcation. Moreover, you should use container-managed transaction demarcation because it is less prone to error, and you should let the container handle transaction demarcation automatically. It frees the component provider from writing transaction demarcation code in the component. It is easier to group enterprise beans to perform a certain task with specific transaction behavior. The application assembler can customize the transaction attributes in the deployment descriptor without modifying the code.

However, programmatic (procedural) access control is sometimes necessary to satisfy fine-grained or application-specific conditions.

Bean Managed Vs Container Managed Security

To simplify the development process for the enterprise bean provider, the implementation of the security infrastructure is left to the EJB container provider and the task of defining security policies is left to the bean deployer. By avoiding putting hard-coded security policies inside bean code, EJB applications gain flexibility when configuring and reconfiguring security policies for complex enterprise applications. Applications also gain portability across different EJB servers that may use different security mechanisms. EJB framework specifies flexibility with regard to security management, allowing it to be declarative (container-managed) or programmatic (bean-managed).

Security management that defines method permissions is declared in the enterprise bean's deployment descriptor or by using annotations (if using EJB 3.0). Container-managed security makes an enterprise bean more flexible, since it isn't tied to the security roles defined by a particular application. A security role is a name given to a grouping of information resource access permissions that are defined for an application. Associating a principal with this security role grants the associated access permissions to that principal as long as the principal is in the role.

However, programmatic (procedural) access control is sometimes necessary to satisfy fine-grained or application-specific conditions. Enterprise beans can programmatically manage the security by using the *isCallerInRole*() and *getCallerPrincipal*() methods contained on the EJBs context object. The *isCallerInRole*() method tests whether the caller has a given security role, returning true if the caller has and false if not. The *getCallerPrincipal*() method returns the java.security.Principal that identifies the caller.

QoS Considerations

• Performance – A measure of the system in terms of response time or number of transactions per unit time. Load Distribution (e.g. DNS Round Robin) and Load Balancing are two techniques that aid in higher performance. Tasks such as Application Tuning, Server Tuning, and Database Tuning also improve system perform.

o DNS Round Robin: A process for distributing load in a system. If we have ten web servers that can service HTTP

requests, the first request is directed to server 1, the second to server 2 and so on. When all ten servers have serviced one request each, the process starts all over again. Note that this one of the load distribution technique. DNS Round Robin does not balance the load.

o Reverse proxy load balancing: Reverse proxy load balancing is used when you have servers with different amounts of CPUs and Memory. You might have some powerful servers just to be used for SSL sessions and others to handle static html. Using this will maximize the performance of your application

• Scalability – The ability of a system to perform and behave in a satisfactory manner with increases in load. Scalability can be achieved in two ways – Vertical (adding additional processors, memory or disks to existing hardware) and Horizontal (adding more machines to the system.) Vertical scalability is easier to implement than Horizontal scalability. Many J2EE vendors do however support horizontal scaling as well.

• Reliability – The ability of a system to assure the integrity and consistency of the application and all its data as the load increases.

• Availability – The ability of a system to assure that all services and resources are always accessible. This can be achieved through fault tolerance (the ability to prevent system failures in the event of service(s) / component(s) failures, commonly implemented via redundancy) techniques such as Active and Passive Replication.

Legacy Connectivity Options

• Screen Scraper: A screen scraper emulates a mainframe terminal. Basically the screen scraper logs onto the mainframe like a normal user and sends requests to the mainframe and then reads the response. The problem with a screen scraper is that if you change any of the mainframes code there is always the possibility that the screen scraper will stop working.

• Java Native Interface: JNI is used to allow Java to communicate with programs written in languages like C++. In effect you are wrapping the C++ code to make it available to Java. For example you will wrap a C++ method *debitAccount (int amount)* with a similar Java method; the Java method will just call the C method. This means you can now make the method accessible via RMI

• JAVA IDL: Java IDL adds CORBA (Common Object Request Broker Architecture) capability to the Java platform, providing standards-based interoperability and connectivity. Java IDL enables distributed Web-enabled Java applications to transparently invoke operations on remote network services using the industry standard IDL (Object Management Group Interface Definition Language) and IIOP (Internet Inter-ORB Protocol) defined by the Object Management Group. Runtime components include Java ORB for distributed computing using IIOP communication.

Hibernate Vs JPA

JPA is just a specification, meaning there is no implementation. You can annotate your classes as much as

you would like with JPA annotations, however without an implementation nothing will happen. Think of JPA as the guidelines that must be followed or an interface, while Hibernate's JPA implementation is code that meets the API as defined by the JPA specification and provides the under the hood functionality.

When you use Hibernate with JPA you are actually using the Hibernate JPA implementation. The benefit of this is that you can swap out Hibernate's implementation of JPA for another implementation of the JPA specification. When you use Hibernate you are locking into the implementation because other ORMs may use different methods/configurations and annotations, therefore you cannot just switch over to another ORM. As an architect you have a choice of JPA implementations for e.g Oracle provides its JPA reference implementation which is EclipseLink

JPA Pros:

• Isolate application from the database.

• Increase manageability of application.

• Improve developer productivity.

• Ease of development generated correct and efficient persistence logic.

• Can outperform hand-crafted SQL, reducing the number of database round-trips.

• JPA entities can be used outside container environment hence promoting re-usability and coding effort.

• JPA entities can be used on both presentation layer and business logic layer, thus reducing coding effort, size of code and maintenance cost associated

• Supports lazy loading of objects hence improving performance.

• Best integration with EJB 3.

• Used when model has complex relationships between tables.

JPA Cons:

• Imply loss of performance cause include additional layer.

• Can not explicitly tunning SQL by hand.

• When datastore is not well supported by ORM providers.

• Developers need skilled in ORM (learning curve).

Java EE Vs Spring

Some points to help you choose between Java EE 6 and Spring

• Freedom to choose container - There are more than 14 Java EE 6 compliant application servers today, with a variety of open source and commercial offerings. A Java EE 6 application can be deployed on any of those containers. So if

you deployed your application on GlassFish today and would like to scale up then you can deploy the same application to WebLogic. And because of the portability of a Java EE 6 application, you can even take it a different vendor altogether. Spring requires a runtime which could be any of these app servers as well. Spring also has a different definition of portability where they claim to bundle all the libraries in the WAR file and move to any application server. But that archive could be bloated.

• Vendor choice - The Java EE 6 platform is created using the Java Community Process where all the big players like Oracle, IBM, RedHat, and Apache are contributing to make the platform successful. Each application server provides the basic Java EE 6 platform compliance and has its own competitive offerings. This allows you to choose an application server for deploying your Java EE 6 applications. If you are not happy with the support or feature of one vendor then you can move your application to a different vendor because of the portability promise offered by the platform.

• Spring is a set of products from a single company, one price book, one support organization, one sustaining organization, one sales organization, etc. Java EE, backed by multiple vendors, is a safer bet for those that are risk averse.

• Production support - With Spring, typically you need to get support from two vendors - VMWare and the container provider. With Java EE 6, all of this is typically provided by one vendor. For example, Oracle offers commercial support from systems, operating systems, JDK, application server, and applications on top of them. VMWare certainly offers

complete production support but this will putting all your eggs in one basket.

• Maintainability - With Spring, you will be building your own distribution with multiple JAR files, integrating, patching, versioning, etc of all those components. Spring's claim is that multiple JAR files allow you can pick the latest versions of different components. The Java EE application servers manage all of this for you and provide a well-tested and commercially supported bundle.

While it is always good to realize that there is something new and improved that updates and replaces older frameworks like Spring, the good news is the Java EE 6 container not only offers what is described, but also lets you deploy and run your Spring applications on them while you go through an upgrade to a more modern architecture. The end result is you get the best of both worlds keeping your legacy investment but moving to a more agile, lightweight world of Java EE 6.

Figure 3: Java EE Vs Spring

JSF Vs JSP

Using JSF, one will get automatic binding of data to visual components, mapping, and validation of input data and saving of state between requests. Using JSP - you have to implement it manually. JSF also supports Ajax interaction with the server.

If you are developing an application that requires a standard behavior, such as data entry, editing and display - JSF is the best bet. If you need an online application with fast response, navigation within a page on the client side, such as Gmail or twitter - you can spend a lot of time translating the client-side logic components in the JSF. In this case, you better use the JSP, as a source of data and JavaScript library to display data. Therefore JSF is not suitable for all types of projects so those who use JSP will not be moving to JSF.

JSF Pros :

• MVC framework. Offers a clean separation between behavior and presentation.

• UI component model (binding, events, state saving, validation).

• Multiples front-ends (desktop and mobiles browsers).

• Hides the HTTP infrastructure.

• Use IDE RAD environments.

• Automatic event handling (map http requests to component specific).

• Automatic server side validations and conversion.

• Automatic I18N e I10L.

• All features result in reduces development time.

• Expression Language and Tag Libraries

JSF Cons :

• Standard HTML editor will not render the JSF. Need to deploy JSF app to page rendered.

Estimating Resource Needs – Hardware Sizing

The NFR specifies the no. of concurrent users and the required up time. The transactions per second – tps can be derived from this data. So below are a few examples that can give you an idea for the hardware sizing of the solution.

Example 1: Business Application

The application is a JEE learning management software that must support 2000 concurrent users with the availability of 24/7 and hence needs a very powerful DB Hardware. The infrastructure is:

• 2 physical Application Server nodes and HTTP Server

o 4 GB Memory (2GB will be for OS)

o 8 (virtual) processors

• Database

o master-slave with automatic fail-over

o 32 GB RAM

o 24 (virtual) processors

Example 2: Web Application

Liferay Portal for 5000 registered users with 500 of them accessing it concurrently would need an application server with 4GB memory and a 3GHz quad-core Xeon processor; for the database, a server below 4GB/1-quad 3GHz would suffice.

Conclusion

Given the examples above, other information, for a requirement of 200 concurrent users doing no resource-intensive actions and 10h/day availability, One 4GB, 2-core 2.5GHz machine running both a single AS and the DB would be able to satisfy the performance requirements.

IBM recommends typical deployments of IBM WebSphere Application Server v7.0 require at least a high-cpu medium instance to have access to enough physical memory and computing power. That means 2 GB RAM and 2 (virtual) cores each with the power of 2.5-3.0 GHz 2007 Opteron or Xeon processor. This is minimum production configuration to run WebSphere and can be taken is as the minimum configuration for any standard Java EE web application.

Often installations find that the database server runs out of capacity much sooner than the WebLogic Server does. You must plan for a database server that is sufficiently robust to handle the application. A good application will require a database three to four times more powerful than the application server hardware.

Multiple Nodes Vs Single Node in Production

It will be sufficient to deploy a single server instance on a machine for limited number of concurrent user's e.g 400 concurrent users, since the Application Server and accompanying JVM are both designed to scale for multiple processors. However, it can be beneficial to create multiple instances on one machine for application isolation and rolling upgrades.

Deployment Environment – Cloud or Physical

You may choose one of three deployment environments:

• Renting a **physical server** – Is the most expensive but may be more cost-effective in the long term, especially if there is permanent high load on the server.

• Renting a **Virtual Private Server** (VPS) –Offerings have low resources (such as 0.5-1.5 GB, ~ 1 GHz) and are thus rarely suitable for Java EE

• **Cloud Deployment** – very flexible (easy addition of a new instance, scale an instance up/down) will provide very good SLAs, but the final price may increase due to various small fees that gets accumulated.

Summary

Remember a clear and easy-to-understand picture is worth a thousand words. It is important to remember that the diagrams/views provided represent one way of documenting the java solution. As long as you follow the criteria outlined in the earlier sections, feel free to provide your own UML diagrams that showcase your unique solution to the assigned business scenario. Specifically do the following:

• Emphasis should be on the business solution i.e. if your class diagram is 80% frameworks and 20% business logic and is a candidate for failure.

• Augment diagrams with English text, but ensure that your diagrams hold water on their own. Make sure the views/ diagrams are legible.

• Common parts of an enterprise web application may not be mentioned explicitly in the requirements like user registration/login, administration. This should be addressed by the solution.

• Engage with stakeholders and conduct workshop and discussions if there is no clarity around certain aspects of the requirements or what is expected to be delivered as part of the solution.

• Make suitable assumption when required and document these in the specifications so that that is available for all parties.

• Risk and Mitigation is an important area and must be covered. All the high-level, systemic risks should be addressed and mitigation actions should be identified.

• Design decisions should be documented along with the rationale in the design artifact.

Business Problem – Case Study

Background:

The Oil and Gas industry is under constant pressure to innovate. Geopolitical instability, environmental , rising energy prices and consumption have created a business environment where inflexible oil companies are penalized by a combination of free market pricing and government sanctions, but where nimble and flexible companies can reap monetary rewards. Zamco is an oil company headquartered in the United Kingdom, with important operations in Middle-east and the Gulf of Mexico. The company has recently entered into an agreement with a large oil auctioning marketplace to bring its product to market more efficiently, improving cash flow and reduce the cost of distribution. The internal business case at Zamco projects cost savings of 22 percent over five years and improved margin and yield management of 6 percent per annum over the current steady state of the business, making this one of the most important projects currently under development at the company.

Workshop Output

You are the architect for the marketplace project at Zamco. You have been tasked by the Zamco management to lead the team responsible for the design, implementation and ongoing management of the complete system as a turnkey or complete solution. After an intensive series of discovery workshops with in-house business analysts and subject matter experts, you know the following facts:

• The oil auctioning marketplace exposes a Java technology–based API using Java Message Service (TMS) technology to allow companies to send messages placing oil for sale and to bid for oil with characteristics, such as guaranteed messaging, message acknowledgment, and message security available.

• Zamco already runs a complex pricing system in-house that calculates what the price for each placement of oil in the auction marketplace should be (based on how it was extracted, its distance and transportation method to its final destination point). As the solution architect, you need to integrate with this system using web services to price the oil before making it available to the auction market place for sale.

• The final remaining external system is Zamco' s inventory management system that is also accessed using web services and will allow you to see what unsold capacity remains available for auction.

• Zamco has a relationship with Merchant Bank to handle the transactions.

• The actual placement of orders for the auction management system are handled manually Zamco employs a team of traders who track the rise and fall of oil prices and use a combination of timing and pricing information and other systems outside the scope of this project to determine the best time to buy or sell oil placements.

• System performance (99 percent of all messages to be constructed and sent in three seconds or less to the IP address

of the API server), scalability (400 concurrent users), availability (99.999 percent during core working hours), and security (128-bit encryption at a minimum) are all key requirements and you must explicitly address each requirement in your proposed solution.

Domain Model: The following diagram depicts the domain model for the business scenario.

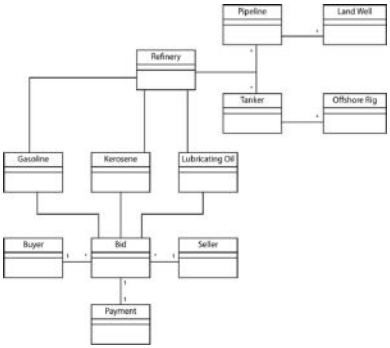

Figure 4: Zamco Domain Model

Use Cases: The following diagram depicts the business use cases for the solution scenario.

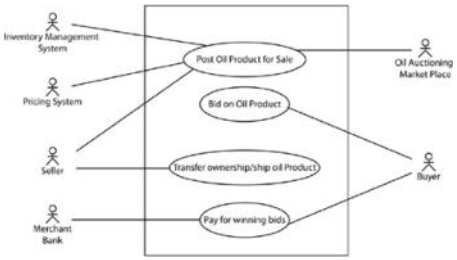

Figure 5: Use Case Diagram Zamco

Solution

This document provides comprehensive architectural overview of Market Place Integration System for Zamco. This application will be referred to as Zamco Application in this artifact aka System Under Development - SuD. It serves as a communication medium between the Software Architect and other project team roles regarding System Architecture. Zamco Application will be developed using a Java EE architecture framework and component-based architecture. This solution architecture document will address the interlinking of various components/sub-systems that participate in the end to end interactions to achieve the business goals and objectives. This architecture adheres to the Java EE 5 standards and best practice architectural guidelines and methodologies and elaborates them in the context of current integration project.

The purpose of the solution architecture document is to provide:

• An end to end overview to various stakeholders regarding the functionality to be implemented as part of the scope

• A link between the business scenarios in terms of how they are interlinked to derive a tangible e2e functionality for the Zamco business

• Key architecture decisions and assumptions

• The Risk and mitigation actions for Zamco Application

• This document provides the following UML views

o Class Diagram

o Component Diagram

o Deployment

o Sequence Diagram

• Overview of non-functional aspects for the solution and how they are addresses in the solution.

Objective and Scope

This section describes the scope and objective for the Zamco Market Integration Project. The Oil and Gas industry is under constant pressure to innovate. Geopolitical instability, environmental issues, and rising energy prices and consumption have created a business environment where inflexible oil companies are punished by a combination of free market pricing and government sanctions, but where nimble and flexible companies can reap monetary rewards. Zamco is an oil company headquartered in the United Kingdom, with important operations in Russia, Africa, and the Gulf of Mexico. The company has recently entered into an agreement with a large oil auctioning marketplace to bring its product to market more efficiently, improving cash flow and reduce the cost of distribution. The internal business case at Zamco projects cost savings of 22 percent over five years and improved margin and yield management of 7 percent per annum over the current steady state of the business

Key Use Case that are in the scope of this project:

• Post Oil Product for Sale

• Bid on Oil Product

• Transfer Ownership/Ship Oil Product

• Pay for Winning Bids

The system designed must meet the following criteria i.e. non-functional requirements

• Performance: 99% of all transactions to be under 3 seconds or less.

• Scalability: Should support 400 concurrent users

• Availability: 99.999% availability during core working hours.

• Security: A Minimum of 128 bit encryption.

Zamco Business Domain Model

The following domain model describes the key objects identified during the workshops for Zamco Market Integration Project. These objects and relationships are addressed in the solution design. Pl refer to domain model in figure 4.

Assumptions

#	System/ Component/ Framework	Assumption
1	Auctioning Market Place	The auctioning market place will provide the capability / functionality for auctioning the products. The Zamco Application - SuD (System under Development) will interface with this external system thro' the published interfaces.
2	Auctioning Market Place	The auctioning market place interfaces will provide the requisite mechanism for de-cryption/encryption, access control and transport layer security.
3	Auctioning Market Place	The standard SLA (response time) for auctioning market place interface will be fewer than 2 seconds
4	Merchant Banking System	The Merchant Bank will provide an open standards interface for the Zamco Application – SuD to be consumed that will be based on interoperable protocol like SOAP over HTTP
5	Merchant Banking System	The Merchant Banking System interface will provide a mechanism for de-cryption/ encryption, access control and transport layer security.
6	Merchant Banking System	The standard SLA (response time) for merchant bank interface will be fewer than 2 seconds
7	Inventory System	The standard SLA (response time) for Inventory System interface will be fewer than 2 seconds

8	Pricing System	The standard SLA (response time) for Pricing System interface will be fewer than 2 seconds
9	Pricing System	Zamco Application integrates with Pricing Systems for pricing the Oil Product before it's placed in the auction market place. The pricing system will invoke the requisite external system for pricing calculation.
10	Deployment Configuration	Web Server, Application Server and DB Server details are not included in the requirements and will be finalized with Zamco.
11	Zamco Application JEE Version	Architecture will follow the JEE 5 specification and hence JPA and JTA will be leveraged for persistence and transactions mechanism.
12	Zamco Application Security	The application will use LDAP for authentication at the web tier. – The application will use role based security at the web and business tier for authorization.

Architecture Decisions

#	System/ Component/ Framework	Architecture Decision
1	Zamco Architecture Framework	Consistent structure and functional separation imposed by the MVC framework makes the applications more reliable and easier to maintain and extend. For these reasons, the

		Zamco application is designed using framework.
2	Common Capabilities	Application will be built using error & exception handling, logging and resource injection capability provided by the JEE framework. This will make the application maintainable, reliable and facilitate faster issue resolutions.
3	Design Patterns	Design Patters are leveraged thro' out the solution architecture as they make the application maintainable and extensible.
4	Java EE Framework	To provide scalability and business logic reusability EJBs are leveraged to build the solution. Enterprise Beans technology provides scalability, reliability, a Component-Based development model, and common horizontal services such as pooling, declarative transaction and security.
5	Distributed Architecture Support	The Zamco Application is designed with open standards based and interoperable interfaces and components for supporting distributed architecture.

Risks and Mitigation

Risk is the product of probability of a threat exploiting vulnerability and the impact to the organization. The process of architecture risk management is the process of identifying those risks in software and then addressing them.

Risk mitigation refers to the process of prioritizing, implementing and maintaining the appropriate risk reducing

measures recommended from the risk analysis process. Mitigating a risk means changing the architecture of the software or business in one of more ways to reduce the likelihood or impact of risk.

#	Risk	Mitigation
1	Lack of Security Mechanism/ controls while interacting with the Merchant Bank will result in the sensitive data being compromised	Merchant Bank architecture should provide encryption/ decryption components while interacting with the Bank APIs
2	Lack of Security Mechanism/ controls while interacting with the Auctioning Market Place will result in the sensitive data being compromised	Auctioning Market Place architecture should provide encryption/deprecation components while interacting with the Auctioning Market Place
3	Application/Interface non-availability for Merchant Bank will result in decrease of customer satisfaction resulting in loss of revenues	Merchant Bank should provide high availability architecture to ensure that the Zamco NFRs are met.
4	Application/Interface non-availability for Auctioning Market Place will result in decrease of customer satisfaction resulting in loss of revenues	Auctioning Market Place should provide high availability architecture to ensure that the Zamco NFRs are met.
5	Application/Interface non-availability for Pricing System will result in decrease of customer	Pricing System should provide high availability architecture to ensure that NFRs are met.

	satisfaction resulting in loss of revenues	
6	Application/Interface non-availability for Inventory System will result in decrease of customer satisfaction resulting in loss of revenues	Inventory System should provide high availability architecture to ensure that NFRs are met.
7	Lack of Zamco Applications Scalability will result in decrease of customer satisfaction	Java EE stateless session beans are used for handling core business logic to support high scalability and distribution while clustering the server. Load balancer can be configured to route traffic to the backup servers when primary's reaches its capacity threshold.

Zamco Architecture Overview

Zamco Application has its components grouped into five major tiers. They are Client Tier, Presentation tier, Business Tier, Integration Tier, and Data Tier. The System is accessed using internet browsers like Internet explorer and Firefox over internet or intranet.

• The Client Tier is the point from where the users connect to the Application. Zamco Application uses Web browsers to access the Application.

o Browsers

• The Presentation Tier is used to host the presentation components and manage session for each user. Zamco Application Web container and Web components are grouped in this tier.

o JSF Framework

• The Business Logic Tier is where the business application processing takes place. Zamco Application uses EJB Container and Business components are grouped in this tier.

o EJBs (Entity's , Session and Message Driven Bean)

o Java Mail

o JAAS

o JAXP

o JAXB

• The Integration Tier handles connection to EIS, retrieving and updating data into the EIS tier. Entities are used for search and large record retrieval.

o JMS/MQ (Encryption/Decryption)

o JAX-WS /Web Services (Encryption/Decryption)

• The Data Tier is where the application's data is persisted in relational databases or legacy application. Zamco Application uses RDBMS for managing the data.

o JPA

o JTA

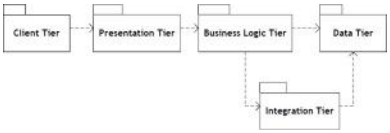

Figure 6: Application Tiers – Zamco

The solution architecture defines the key system, application components and integration necessary to process the data and support Zamco business processes.

Zamco Application Framework

This section provides the details of the framework components and corresponding design patterns leveraged in the application architecture.

PRESENTATION REQUEST PROCESSING

Presentation Request Processing has been derived by combining the following:

• Front Controller

• Command Handler (BackingBean Classes)

• Business Delegate (Business Service Provider/Proxies to Business Services)

The participant diagram and the collaboration diagram of the mechanism are shown below.

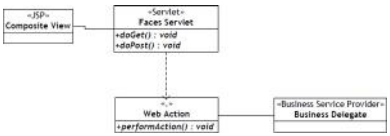

Figure 7: Presentation Request Processing – Class Diagram

The following key architectural decisions have been captured by the Presentation Request Processing mechanisms:

• All user requests are handled by a single web Controller (the front controller of the application)

• The front controller forwards the request to the appropriate Command Handler with the help of resource and access mapping information stored in the form of an XML file (BackingBean.xml)

• Command Handler manages user interactions for a specific use case and co-ordinates use case interactions with the user.

• The Command Handlers does not perform business logic. Instead, they use Business delegates as the Business provider to the Application Server services.

• The Command Handlers do not produce any User output. Instead, they call views to create HTML type of a stream that is returned to the user.

• Views are implemented as JSPs.

• Business delegates are implemented as Java Beans that run in the thread of the calling Delegates.

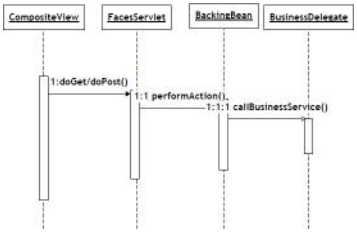

Figure 8: Presentation Request Processing – Sequence Diagram

BUSINESS SERVICE ACCESS

Business Service Access combines implementation strategies of the following Sun Java EE patterns:

• Business Delegate

• Session Facade

• Service Locator

The participant diagram and the collaboration diagram of the mechanism are shown below.

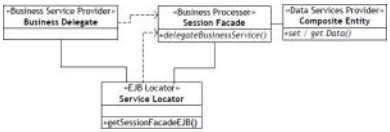

Figure 9: Business Service Access – Class Diagram

This mechanism is a blueprint to access the Application Server components for the organisation. The architecture does not allow any Presentation layer components (Command Handlers in particular) to communicate directly with entities. Hence only beans that can be accessed remotely are session beans.

The mechanism captures the following architectural decisions:

• All business service components are implemented as Session EJBs or have Session EJB facades.

• A client (in most cases it is a Bean handlers) that requires access to a Business service component and creates an instance of a Business Delegate.

As a consequence of using two mechanism of framework, the application has a distinct separation between presentation and business logic. The boundary is business delegates, which are smart proxies to business services of application. This separation is desirable for few reasons.

• It provides an explicit contract between designers and developers of presentation tier and the business tier of the system

• It allows the business tier to change independently from presentation tier

• It allows for concurrent development of the two tiers.

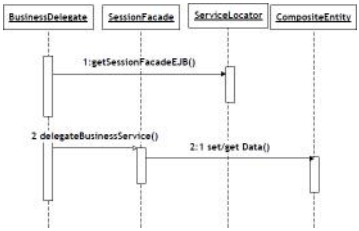

Figure 10: Business Service Access – Sequence Diagram

DATA SERVICE ACCESS

The application architecture uses Local CMP Entity Beans for read/write operations on the database. These will handle concurrency for accounts when there are situations both customer and agents accessing the same details.

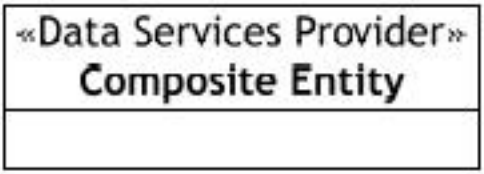

Figure 11: Data Service Provider – Zamco

COMMON ELEMENT & SERVICES

Common elements and services is the grouping (a package) of support elements and services that do not belong to any of the Business components, but belong to the framework that underlines the Application.

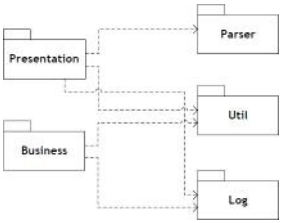

Figure 12: Common Services – Zamco

The package contains the following elements and services:

• Presentation Request Processing

• Business Request Processing

• Logging Framework

• Utility Classes

• XML Parser

Zamco UML Diagrams

The following section depicts the various diagrams for the Zamco application. This section consists of the following diagrams

• Class Diagram

• Component Diagram

• Deployment Diagram

• Sequence Diagrams

This section along with application framework forms the end-2-end depiction of architecture for Zamco application. The application framework section predominately focused on the presentation and business tier consisting of the MVC framework and the various classes that participate in MVC architecture e.g business delegate, front controller, service locator and business services EJBs. There is also a master list of all classes for Zamco Application that is shown as part of the List of Class Zamco section.

MAIN CLASS DIAGRAM

This section depicts the main class diagram for the Zamco Application. This diagram depicts the key classes for Zamco application. The class diagram is developed from the domain model and models elements as entities, session beans, MDB and Java Bean, as well as reorganizing the elements into a classic n-tier layered architecture. The section List of Classes – Zamco provides the comprehensive list of all the classes for the Zamco application along with the description. This is the master list of all classes in the Zamco Application. The main class diagram shows details pertaining the following:

• JSPs for the Key Use Case Scenarios

• Controller/s

• Entities

• Stateless Sessions Beans (Business Logic)

• Message Driven Beans

• External Application/Systems Interface

Classes omitted (to ensure readability) from the main class diagram and are covered as part of the application framework section or other diagrams.

• JSPs (Login, Logout, Home) & (Admin)

• Backing beans for All JSPs

• Business Delegates

• Service Locator

The class diagram remains framework agnostic and any web framework is an acceptable choice. The Zamco Application leverages JSF for the presentation tier design. The class diagram uses annotations to show how the domain model have been mapped to JEE components specifically sessions beans and entity beans.

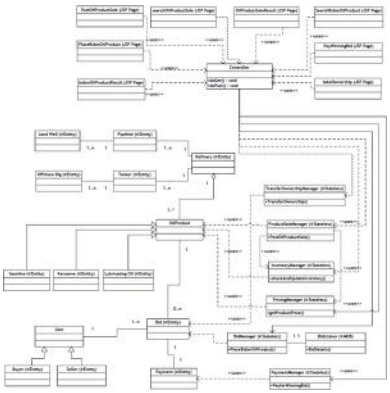

Figure 13: Main Class Diagram – Zamco

WEB TIER CLASS DIAGRAM

The following depicts the web tier class diagram for Zamco Application.

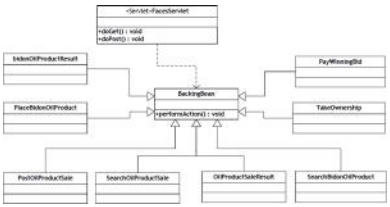

Figure 14: Web Tier Class Diagram – Zamco

LIST OF ALL THE CLASSES

This section provides comprehensive list of all the classes for the Zamco application. This is master list of all the classes in the Zamco Application. This table provides the class name, its tier, its sub-system and the description for each in the below table.

#	Tier	Sub System	Class	Description
1	Presentation Tier	Oil Product JSPs	SearchOilProductSaleJSP. OilProductSaleResultJSP, PostOilProductSaleJSP	JSPs for Post Oil Product for Sale use case
2	Presentation Tier	Bid JSPs	SearchBidonOilProductJSP, bidonOilProductResultJSP, PlaceBidonOilProductJSP	JSPs for Bid on Oil Product use case
3	Presentation Tier	Take Ownership JSPs	takeOwnershipJSP	JSP for Take Ownership use case
4	Presentation Tier	Pay Winning Bid JSPs	PayWinningBidJSP	JSPs for Pay Winning Bid use case
5	Presentation Tier	Login, Logout, Home JSPs	LoginJSP, LogoutJSP, HomeJSP	JSPs for logging in, logging out and Home Use Case
6	Presentation Tier	Admin JSPs	UserJSP, BidJSP, ProductJSP, PaymentJSP	JSPs for administration

7	Presentation Tier	Front Controller	FacesServlet	Front Controller for the Zamco Application which will route the request to appropriate Web Bean Classes
8	Presentation Tier	Oil Product Backing bean	SearchOilProductSale, OilProductSaleResult, PostOilProductSale	This is routed from the Front Controller. This invokes the business service via ProductSaleManagerBD (business delegate)
9	Presentation Tier	Bid Backing Bean	SearchBidonOilProduct, bidonOilProductResult, PlaceBidonOilProduct	This is routed from the Front Controller. This invokes the business service via BidManagerBD (business delegate)
10	Presentation Tier	Take Ownership Backing bean	TakeOwnership	This is routed from the Front Controller. This invokes the business service via TransferOwnershipManagerBD (business delegate)
11	Presentation Tier	Pay Winning Bid Back bean	PayWinningBid	This is routed from the Front Controller. This invokes the business service via PaymentManagerBD (business delegate)
12	Presentation Tier	Login, Logout, and Home Back bean	Login,Logout,Home	This is routed from the Front Controller. This invokes the business service via UserManagerBD (business delegate)
13	Presentation Tier	User, Bid, Product, Payment Back bean	User,Bid,Product, Payment	This is routed from the Front Controller. This invokes the business service via

				AdminManagerBD (business delegate)
14	Presentation Tier	Oil Product Business Delegate	ProductSaleManagerBD	This is business service proxy for ProductSaleManager. It hides JNDI lookup and remote exception log from web bean classes
15	Presentation Tier	Bid Business Delegate	BidManagerBD	This is business service proxy for BidManager. It hides JNDI lookup and remote exception log from web bean classes
16	Presentation Tier	Take Ownership Business Delegate	TransferOwnershipManagerBD	This is business service proxy for TransferOwnershipManager. It hides JNDI lookup and remote exception log from web bean classes
17	Presentation Tier	Pay Winning Bid Business Delegate	PaymentManagerBD	This is business service proxy for PaymentManager. It hides JNDI lookup and remote exception log from web bean classes
18	Presentation Tier	Login, Logout and Home Business Delegate	UserManagerBD	This is business service proxy for UserManager. It hides JNDI lookup and remote exception log from web bean classes
19	Presentation Tier	User, Bid Product and Payment Business Delegate	AdminManagerBD	This is business service proxy for AdminManager. It hides JNDI lookup and remote exception log from web bean classes
20	Business	Service	ServiceLocator	Creates, Caches and

	Tier	Locator		Returns Business Services EJBs
21	Business Tier	Stateless Session Bean	ProductSaleManager	Business process component for product sale and acts as a façade for entity layer
22	Business Tier	Stateless Session Bean	BidManager	Business process component for bid management and acts as a façade for entity layer
23	Business Tier	Stateless Session Bean	TransferOwnershipManager	Business process component for transfer ownership and acts as a façade for entity layer
24	Business Tier	Stateless Session Bean	PaymentManager	Business process component for payment and acts as a façade for entity layer
25	Business Tier	Stateless Session Bean	InventoryManager	Business process component for inventory and acts as a façade for entity layer
26	Business Tier	Stateless Session Bean	PricingManager	Business process component for pricing and acts as a façade for entity layer
27	Business Tier	Stateless Session Bean	UserManager	Business process component for authentication/authorization and acts as a façade for entity layer
28	Business Tier	Stateless Session Bean	AdminManager	Business process component for administration of various aspects e.g user, bid, product payment and acts as a façade for entity layer

29	Integration Tier	Message Driven Bean	BidListener	This listens to the incoming BidDetails from the Auctioning Market Place
30	Persistence Tier	JPA Entities	Refinery, Pipeline,Tanker, Landwell, OffshoreRig	JPA entries for Zamco Application
31	Persistence Tier	JPA Entities	Gasoline, Kerosene, Lubricating Oil	JPA entries for Zamco Application
32	Persistence Tier	JPA Entities	Buyer, Seller	JPA entries for Zamco Application
33	Persistence Tier	JPA Entities	Bid	JPA entries for Zamco Application
34	Persistence Tier	JPA Entities	Payment	JPA entries for Zamco Application
35	Integration Tier	Value Objects	RefineryVO, PipelineVO,TankerVO, LandwellVO, OffshoreRigVO, GasolineVO, KeroseneVO, LubricatingOilVO, BuyerVO, SellerVO, BidVO, PaymentVO	Value Objects

COMPONENT DIAGRAM

The section depicts the component diagram for the Zamco Application. This diagram shows the component interactions for the Zamco Application. The component diagram is another view of the system, at a higher level than the class diagram. The component diagram builds further on the class diagram, related classes are logically grouping together into components that carry out a distinct business operation. The component diagram is laid out so as to make the layered nature of the architecture clear.

This component diagram consist of the following tiers

• Presentation Tier

• Business Logic & Persistence Tier

• Integration Tier

• External Systems / Interfaces

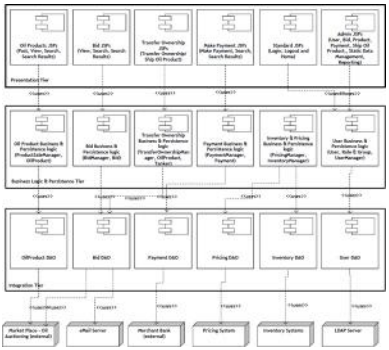

Figure 15: Component Diagram – Zamco

Deployment Diagram!

The following diagram shows the deployment diagram for the Zamco Application. The deployment diagram captures information about how the system operates in production. The deployment diagram depicts how the proposed solution will execute at runtime, including the hardware and configuration that will be needed to support the software solution. In this diagram, we have adopted the convention of specifying three hardware profiles (A, B and C) to call out the fact that we expect the web server, application servers and the database server to require different system resources.

The deployment diagram is divided into the following tiers:

• Web Server

• Application Server

• Database Server

• External Applications

To be able to provide high availability a hot backup has been configured. Although primary application server is capable of handling 400 users, hot backup server will be able to handle users when primary server fails. Load balancer will be configured to route traffic when primary fails or reaches its capacity threshold.

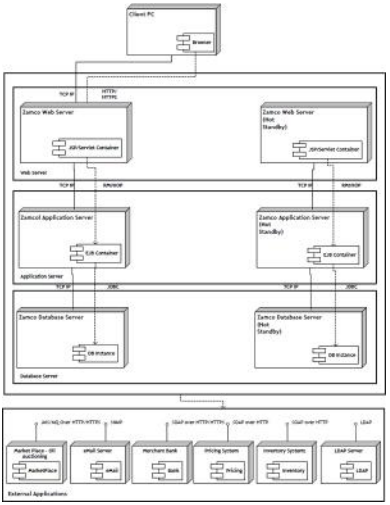

Figure 16: Deployment Diagram – Zamco

NODE DESCRIPTIONS

The following table lists the various nodes that are part of the architecture and the description.

#	Node	Description

1	Client PC	Client PC provides the web browser to access the Zamco application over internet
2	Zamco Web Server	Zamco Web Server hosts the HTTP listener process
3	Zamco Application Server	Zamco Application Server node runs the business components and handles connection pooling
4	Zamco Database Server	Runs the RDBMS instance
5	Web Container	Java EE Servlet Container provided by web server. All servlets and JSPs will be executed under control of this Container. In Web Container there will be one Servlet container per Web Application instance. Protocol : HTTP/HTTPS
6	EJB Container	Java EE EJB Container provided by application server. All EJBs will be executed under control of this Container. There will one EJB container per Application instance. Protocol : RMI-IIOP, JMS/MQ, JAX-WS

HARDWARE PROFILES OF NODES

In the deployment diagram, we have adopted the convention of specifying three hardware profiles (A, B and C) to call out the fact that we expect the web server, application servers and the database tier to require different system resources.

#	Tier	Hardware Profile Details

1	Web Tier	CPU: 1 Multi Core CPU, 2.5-3 GHz Memory: 4 GB Disk: 12 K RPM HD OS: 32 Bit Network: 10 GbE
2	Application Tier	CPU: 1 Quad Core CPU, 2.5-3 GHz Memory: 8 GB Disk: 15 K RPM HD OS: 32 Bit Network: 10 GbE
3	Database Tier	CPU: 2 Quad Core CPU, 2.5-3 GHz Memory: 32 GB Disk: 15 K RPM HD OS: 64 Bit Network: 10 GbE

Sequence Diagram

The sequence diagram shows the flow of messages between different objects. The sequence diagram explains the use case realization of Post Oil Product for Sale Use Case, Bid on Oil Product Use Case, Transfer Ownership Use Case, and Pay for Winning Bid Use Cases.

POST OIL PRODUCT FOR SALE – SEQUENCE DIAGRAM

The following diagram shows the mapping to the Post Oil Product for Sale Use Case. This allows the Zamco Representative to place oil product for sale in the market place.

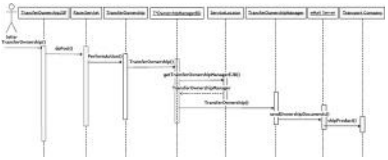

Figure 17: Post Oil Product for Sale Sequence Diagram – Zamco

BID ON OIL PRODUCT – SEQUENCE DIAGRAM

The following diagram shows the mapping to Bid on Oil Product Use Case. This allows potential buyer to place bids for a specific quantity of Oil Product. The potential buyer searches offers matching desired criterion and the system responds with matching offer. The potential buyer selects a specific offer of interest and places bid for the product.

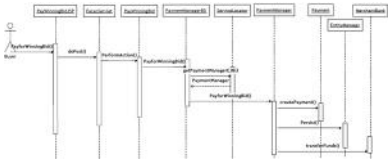

Figure 18: Bid on Oil Product Sequence Diagram – Zamco

TRANSFER OWNERSHIP / SHIP OIL PRODUCT – SEQUENCE DIAGRAM

The following diagram shows the mapping to the Transfer Ownership Use Case. This allows an authorized Zamco Representative to transfer Ownership of the purchased offer directly to the purchaser or to a named intermediary for transport to destination. This is initiated with the Zamco Representative selects a specified close successful offer.

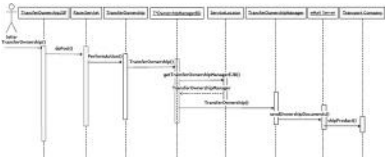

Figure 19: Transfer Ownership Ship Product Sequence Diagram – Zamco

PAY FOR WINNING BID – SEQUENCE DIAGRAM

The following diagram shows the mapping to the Pay for Winning Bid Use Case. This allows the authorized buyer to pay for an oil product won at the auction. After successful buyer executes request for successful bid the system responds with successful bid.

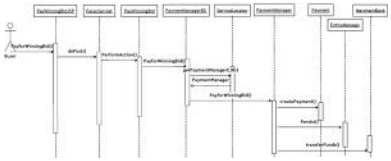

Figure 20: Pay for Winning Bid Sequence Diagram – Zamco

Part II – Non-Functional Requirements

Tackling the NFRs

You should have a clear view on the non-functional characteristics of the application and why your solution meets and or exceeds these requirements. You should also understand and appreciate the main technical risks inherent in both the business and technology landscape and provide mitigations actions.

• Performance

• Scalability

• Reliability

• Security

• Availability

Maintainability Handling NFRs will boil down to the following key aspects:

• Why and How are you meeting the individual non-functional requirements (performance, security, ...)

• What were the individual technological choices you have made and alternatives considered and why you've rejected them.

• Key ones are communication with external systems, Patterns leveraged, frameworks that were leveraged during the design process

NFRs – Solution

The below sections describes the solution options for the NFR for the given business problem.

Performance

The following mechanisms were recommended for Zamco Application architecture to achieve required performance:

• The 3 tiers namely web, application and database are hosted on different hardware machines. Hence the resource intensive operations implemented in EJB tier and the Database are hosted on dedicated hardware providing improved performance.

• Use of connection pooling for data base connections will improve performance.

• Use of bean pooling in the EJB container will improve the performance

• Using Business Delegate pattern will reduce the number of round trips between presentation and Business logic layer hence improving the performance.

• Using Service locator will the improve performance. Service Locator will avoid performance overhead due to initial context creation and service lookups.

• Transfer Object can avoid network performance degradation caused by chattier applications.

• Caching of objects will improve the application performance

Scalability

The following mechanisms were recommended for Zamco Application architecture to achieve required scalability:

• The architecture is designed to support both vertical as well as horizontal scalability.

• To handle more number of client requests the application can be scaled by deploying additional Web containers in multiple machines.

• The application server will be clustered which allows the ability to add processing capability by simply adding instances to the cluster.

• Using connection pooling for database and bean pooling for EJB improves scalability of applications.

• Design of application using stateless session beans improves scalability of application. (ProductSaleManager, BidManager and PaymentManager)

• Application or database servers nodes can be added (horizontal scaling) to improve the scalability of the application.

• The business logic is not tightly coupled with web tier components and hence deploying the application components (EJB components) in separate node and clustering them is very easy.

• The architecture does not use any entity beans (EJB 2.1). This increases the amount of objects created in memory and would enhance scalability.

Availability

The following mechanisms were recommended for Zamco Application architecture to achieve high availability:

• Application is designed with a hot stand-by configuration for high availability.

• Incase the primary servers goes down the load balancer will be able to route the request to secondary / hot standby nodes.

• The load balancer can be configured to route traffic to hot standby in case the primary reaches is capacity threshold.

• Stateless design - When a particular stateless server fails, its work can be re-directed to a different server instance without implications for state management.

Security

The following mechanisms were recommended for Zamco Application architecture to achieve required security:

• The security will be provided through DMZ using inner and outer firewall. Application and Database servers will be behind the inner firewall and the web server will be behind outer firewall in the DMZ.

• The architecture uses Form Based authentication for the web tier and the security logic is implemented in reusable business component (UserManager) in the application tier.

• SSL will provide the desired security for sending sensitive information to critical systems like Merchant Bank and Market Place

• JMS Message-level (Using Encryption and Digital signatures) and Transport-layer security (Using SSL).

• The application will use LDAP for authentication at the web tier and will use role based security at the web and business tier for authorization.

Best Practices:

SQL Injection: Prepared or Dynamic SQL Statements

Cross Site Scripting: JSF Validation, Avoid JavaScript's, and Avoid Frame/iFrames

Denial of Service: Service Request Queue Technique. Limiting the number of Concurrent Request and queuing all excess requests.

Man-in-the-Middle: Using SSL, Avoid Frames/iFrames, Avoid URL Rewriting

Reliability

The following mechanisms were recommended for Zamco Application architecture to achieve reliability:

• Primary application server although serves all the request, a backup server is configured to route traffic if primary goes down.

• Data Integrity is achieved with security controls/ mechanisms which will prevent third party from making any unauthorized access and/or changes to data.

• Database transactions would ensure data integrity as it will be a full commit or a rollback based on success/failure status.

Extensibility and Maintainability

The following mechanisms were recommended for Zamco Application architecture to achieve extensibility and maintainability:

• The logical separation of the application into different tiers (Client, Presentation, Business Logic, Integration and EIS tier) allows a system to be flexible and easily maintainable. When there are any changes in presentation, it will be done at presentation tier and it will not affect the other tiers. If there are business logic changes only EJB components business logic is altered and it will not affect other tiers and vice versa.

• The architecture uses command pattern at the web tier (JSF Backing Bean classes) for handling web events. Therefore, when new functionalities are added to the system, it will not affect the existing system. We can easily create a new web action by developing new BackingBean class and configure it in faces-config.xml file. Even modifying the existing functionality becomes easy by changing the respective Backing Bean classes.

• Object oriented design like encapsulation, inheritance, low coupling and high cohesion are leveraged in application design. So any change to sub systems will have less impact on systems which are using it as long the interfaces remain same.

• Independence of interface from implementation – This mechanism allows architects to substitute different implementations for the same functionality.

• This is supported with good documentation of the application (Architecture diagrams, Interface agreements with external systems, Class diagrams, Sequence diagrams, Coding Guidelines etc)

Manageability

The following mechanisms were recommended for Zamco Application architecture to achieve manageability

• Good logging mechanism to indicate any Fatal or Error conditions

• Leverage Tivoli software to provide extensive monitoring and management capability for the application

• Instrumentation and profiling of code to ensure that it meets the quality thresholds

Sessions (State) Management

The following mechanisms were recommended for Zamco Application architecture for session / state management

• The architecture uses JSF Faces servlet (FrontController), which is the gateway to all the incoming requests to the system. It handles the client session at common place in HttpSession object at the web tier.

• EJB tier is not used for handling the stateful session hence affecting the application's scalability.

• Most of the application server supports HTTP session in memory replication and will be used during the applications clustered failover.

Transaction & Concurrency

The following mechanisms were recommended for Zamco Application architecture for transaction & Concurrency

• The application design incorporates container-managed transaction for database transactions, which allows for simpler, more portable code.

• All transactions related to one component are managed by a single workflow manager object through an EJB. This EJB is a stateless session bean designed using the facade pattern.

• The architecture uses JPA Entity (Payment (@Entity)) to handle the payment transaction. The method is invoked with Requires New transaction attribute set via a stateless session bean.

• Entity instance are based on rows and EntityManager takes care of concurrency.

Persistence

The following mechanisms were recommended for Zamco Application architecture for persistence

• The architecture uses JPA Entities to persist the data into the relational database.

• JPA Entity provides composite view of relational tables to easily manage the persistence and JPA query language provides fast access to the table rows with optimized queries.

• The use of JPA Entities to persist the data increases scalability across multiple nodes.

• JPA improves database portability because of database independent JPA Query language.

Distribution

The following mechanisms were recommended for Zamco Application architecture for distribution.

• The architecture uses remote stateless session beans for accessing the business service from web tier. These components act as the facade to the JPA Entity layer, which are designed with composite model to support container managed relationship.

• This way we can distribute the business components by deploying with multiple machines and they can be accessed by the web tier components from remote. This will increase the application's scalability as well.

Part III – Java EE Best Practices

Best Practices – Java EE Applications

To provide some simple guidance for entering this world, I have compiled this best-of-the-best list of what I feel are the most important and significant best practices for Java EE. In order to avoid omitting critical best practices the list instead is an essential top best practices for Java EE application.

Web Tier

Leverage MVC

Clean separate business logic (Java beans and EJB components) from controller logic (servlets/Struts actions) from presentation (JSP, XML/XSLT). Good layering can cover a multitude of sins.

This practice is so central to the successful adoption of Java EE that there is no competition for the #1 slot. Model-View-Controller (MVC) is fundamental to design a good Java EE application. It is simply the division of labor of your programs into the following parts:

Those responsible for business logic (the Model - often implemented using Enterprise JavaBeans or plain old Java objects). Those responsible for presentation of the user interface (the View). Those responsible for application navigation (the Controller -- usually implemented with Java servlets or associated classes like Struts controllers).

There are a number of problems that can emerge from not following basic MVC architecture. The most problems occur

from putting too much into the View portion of the architecture. Practices like using JSP tag libraries to perform database access, or performing application flow control within a JSP are relatively common in small-scale applications, but these can cause issues in later development as JSPs become progressively more difficult to maintain and debug.

Likewise, we often see migration of View layer constructs into business logic. For instance, a common problem is to push XML parsing technologies used in the construction of views into the business layer. The business layer should operate on business objects and not on a particular data representation tied to the view.

However, just having the proper components does not make your application properly layered. It is quite common to find applications that have all three of servlets, JSPs, and EJB components, where the majority of the business logic is done in the servlet layer, or where application navigation is handled in the JSP. You must be rigorous about code review and refactoring to ensure that business logic is handled in the Model layer only, that application navigation is solely the province of the Controller layer, and that your Views are simply concerned with rendering model objects into appropriate HTML and Javascript.

Just a few years ago, user interface developers for Web applications could choose from servlets and JSPs, struts, and perhaps XML/XSL transformation. Since then, Tiles and Faces have become popular, and now AJAX is gaining a strong following. It would be a shame to have to redevelop an

application's core business logic every time the preferred user interface technology changes.

Prefer JSFs as your first choice of presentation technology

Using JSF, one will get automatic binding of data to visual components, mapping, and validation of input data and saving of state between requests. Using JSP - you have to implement it manually. JSF also supports Ajax interaction with the server.

If you are developing an application that requires a standard behavior, such as data entry, editing and display - that JSF is the best bet. If you need an online application with fast response, navigation within a page on the client side, such as Gmail or twitter - you can spend a lot of time translating the client-side logic components in the JSF. In this case, you better use the JSP, as a source of data and JavaScript library to display data.

Store only as much state as you require

Enable session persistence.

HttpSession are great for storing information about application state. Unfortunately, developers often lose sight of the intent of the HttpSession -- to maintain temporary user state. It's not an arbitrary data cache. Many systems put enormous amounts of data -- megabytes -- in each user's session. Well, if there are 1000 logged-in users, each with a 1 MB HTTP session, that's one gigabyte or more of memory in use just for sessions. Keep those HTTP sessions small and if you don't your application's performance will suffer. Good

rule of thumb is something under 2K-4K. This isn't a hard rule. 8K is still okay, but obviously slower than 2K.

One common problem is in using HttpSession to cache information that can be easily recreated, if necessary. Since the sessions are persisted, this is a very expensive decision forcing unnecessary serialization and writing of the data. Instead, use an in memory hash table to cache the data and just keep a key to the data in the session. This enables the data to be recreated should the user fail over to another application server.

Enable session persistence, if you don't enable session persistence, should a server is stopped for any reason (a server failure or ordinary maintenance); any user that is currently on that application server will lose their session. That makes for a very unpleasant experience. They have to log in again and redo whatever they were working on. If instead, session persistence is enabled, AS will automatically move the user (and their session) to another application server, transparently. They won't even know it happened. This works so well, that we've actually seen production systems that crashes regularly still provide adequate service.

Business Tier

Always use session facades whenever you use EJB components

Use local EJBs when architecturally appropriate.

Using a session facade is one of the best-established best practices for the use of EJBs. In fact, the general practice is

widely advocated for any distributed technology, including CORBA, EJB, and DCOM. Basically, the lower the distribution "cross-section" of your application, the less time will be wasted in overhead caused by multiple, repeated network hops for small pieces of data. The way to accomplish this is to create very large-grained "facade" objects that wrap logical subsystems and that can accomplish useful business functions in a single method call. Not only will this reduce network overhead, but within EJBs, it also critically reduces the number of database calls by creating a single transaction context for the entire business function. This is actually one of the core principles of Service Oriented Architecture (SOA).

EJB specification, provide performance optimization for co-located EJBs. Local interfaces must be explicitly invoked by your application, requiring code changes and preventing the ability to later distribute the EJB without application changes. If you are certain the EJB call will always be local, take advantage of the optimization of local EJBs. However, the implementation of the session facade itself, typically a stateless session bean, should be designed for remote interfaces. This way, the EJB itself can be used remotely by other clients without major breakage to existing business logic.

For performance optimization, a local interface can be added to the session facade. This takes advantage of the fact that most of the time, in Web applications at least; your EJB client and the EJB will be co-located within the same JVM. Alternatively, Java EE application server configuration optimizations, such as WebSphere "No Local Copies," can be used if the session facade is invoked locally but using the

remote interface. However, you must be aware that these alternatives change the semantics of the interaction from pass-by-value to pass-by-reference. This can lead to subtle errors in your code. It is best to use local EJBs, since the behavior is controllable on a bean by bean basis, rather than affecting the entire application server.

Use stateless session beans instead of stateful session beans

This makes your system more amenable to failover. Use the HttpSession to store user-specific state.

A stateful session bean is exactly the same, architecturally, as a CORBA object a single object instance, tied to a single server, which is dependent upon that server for its life. If the server goes down, the object values are lost, and any clients of that bean are thus out of luck.

Java EE application servers providing for stateful session bean failover can work around some issues, but stateful solutions are not as scalable as stateless ones. For example, in Application Server, requests for stateless session beans are load-balanced across all of the members of a cluster where a stateless session bean has been deployed. In contrast, application servers cannot load-balance requests to stateful beans. This means load may be spread disproportionately across the servers in your cluster. In addition, the use of stateful session beans pushes state to your application server, which is undesirable. Stateful session beans increase system complexity and complicate failure scenarios. One of the key principles of robust distributed systems is the use of stateless behavior whenever possible.

A stateless session bean approach is chosen for most applications. Any user-specific state necessary for processing should either be passed in as an argument to the EJB methods (and stored outside the EJB through a mechanism like the HttpSession), or be retrieved as part of the EJB transaction from a persistent store (for instance, through the use of Entity beans). Where appropriate, this information can be cached in memory, but beware of the potential challenges that surround keeping the cache consistent in a distributed environment. Caching works best for read-only data.

In general, you should make sure that you plan for scalability from day one. Examine all the assumptions in your design and see if they still hold if your application will run on more than one server. This rule applies not only in application code in the cases outlined above, but also to situations like MBeans and other administrative interfaces.

Use container-managed transactions.

Leverage two-phase commit transactions work in Java EE and rely on them rather than developing your own transaction management. The container will almost always be better at transaction optimization.

Container-managed transactions (CMTs) provide two key advantages that are nearly impossible to obtain without container support: composable units of work, and robust transactional behavior.

If your application code explicitly begins and ends transactions (perhaps using javax.jts.UserTransaction, or even native resource transactions), future requirements to compose

modules, perhaps as part of a refactoring, often requires changing the transaction code. For example, if module A begins a database transaction, updates the database and then commits the transaction and module B does the same, consider what happens when you try to use both from module C. Module C, which is performing what is a single logical action, is actually causing two independent transactions to occur. If module B were to fail during an operation, module A's work is still committed. This is not the desired behavior. If, instead, module A and module B both used CMTs, module C can also start a CMT (typically implicitly via the deployment descriptor) and the work in modules A and B will be implicitly part of the same unit of work without any need for complex rework.

If your application needs to access multiple resources as part of the same operation, you need two-phase commit transactions. For example, if a message is removed from a JMS queue and then a record is updated in a database based on that message, it is important that either both operations occur -- or that neither occurs. If the message was removed from the queue and then the system failed without updating the database, this system is inconsistent. Serious customer and business implications result from inconsistent states.

We occasionally see client applications trying to implement their own solutions. Perhaps the application code will try to "undo" the queue operation if the database update fails. We don't recommend this. The implementation is much more complex than you initially think and there are many corner cases (imagine what happens if the application crashes in the middle of this). Instead, use two-phase commit transactions. If you use CMT and access to two-phase commit capable

resources (like JMS and most databases) in a single CMT, Application Server will take care of the dirty work. It will make sure that the transaction is entirely done or entirely not done, including failure cases such as a system crash, database crash, or whatever. The implementation maintains transactional state in transaction logs. We can't emphasize enough the need to rely on CMT transactions if the application accesses multiple resources. If the resources you are accessing cannot provide for two-phase commit, then of course you have no choice but to use a more complex approach but you should do everything as possible within your power to avoid this situation.

Take advantage of application server features requiring no code modification

With features such as Application Server caching and the Prepared Statement cache, the performance gains are substantial and the overhead is minimal.

Best practice above states a clear case as to why you should be very prudent in applying application-server-specific features that modify your code. It makes portability difficult and may make version migration challenging as well. However, there are a suite of application-server specific features, in Application Server, that you can and should take full advantage of precisely because they do not modify your code. Your code should be written to the specification, but if you know about these features and how to properly use them you can take advantage of significant performance gains.

For one example of this, in WebSphere Application Server, you should turn on dynamic caching and use servlet caching.

The performance gains are substantial and the overhead minimal, while the programming model is unaffected. The merits of caching to improve performance are well understood. Unfortunately, the current Java EE specification does not include a mechanism for servlet/JSP caching. However, WebSphere Application Server provides support for page and fragment caching through its dynamic cache function without requiring any application changes. The cache policy is specified declaratively and configuration is through XML deployment descriptors. Therefore, your application is unaffected, remaining Java EE specification compliant and portable, while benefiting from the performance optimizations provided from WebSphere's servlet and JSP caching.

The performance gains from dynamic caching of servlets and JSPs can be substantial, depending on the application characteristics. Performance benefits go up to a multiplier of 10 from applying dynamic caching to an existing RDF (Resource Description Format) site summary (RSS) servlet.

For additional performance gains, the WebSphere Application Server servlet/JSP results cache is integrated with the WebSphere plug-in ESI Fragment processor, the IBM HTTP Server Fast Response Cache Accelerator (FRCA) and Edge Server caching capabilities. For heavy read-based workloads, significant additional benefits are gained through leveraging these capabilities.

For another example of the principle take advantage of the WebSphere Prepared Statement Cache when writing JDBC code. By default, whenever you use a JDBC PreparedStatement in WebSphere Application Server, it will

compile the statement once and then place it in a cache that will be reused not just later in the same method where the PreparedStatement is created, but across all points in your program where the same SQL code is used in the same or another PreparedStatement. Saving this re-compilation step can result in a significantly lower number of calls to the JDBC driver and improve the performance of your application. You don't have to do anything special to take advantage of this; just write your JDBC code to use PreparedStatements. By writing your code to use a PreparedStatement instead of a regular JDBC Statement class (which uses purely dynamic SQL) you can take advantage of this performance enhancement while not losing any portability.

One more important area that we see ignored far too often is clustering. Applications need to be designed and delivered to run in a clustered environment. Most realistic environments require clustering for scalability and reliability. Applications that don't cluster lead quickly to disaster.

Log your program state using a standard logging framework

This includes exception handlers. Use a logging framework like JDK 1.4 or above logging or Log4J.

Logging is sometimes the most tedious, undervalued part of programming, but it is the difference between long hours of debugging and going home at a reasonable time. As a general rule of thumb, at every transition point, log it. When you're passing parameters from one method to another method, or

between classes, log it. When doing some transformation on an object, log it. When in doubt, log it.

Once you've made the decision to log, choose an appropriate framework. There are lots of good choices out there but we are partial to the JDK 1.4 trace APIs, as they are fully integrated into the WebSphere Application Server trace subsystem and are standards-based.

Persistent Tier

Favor Conventions over Exceptions

In an ideal world, the default configuration settings would always be exactly what we wanted. Our use of configuration by exception would not require any exceptions to be configured. We can approach this ideal world by minimizing the frequency and severity of our deviations from the assumed configuration. Although there is nothing inherently wrong about providing specific exceptions to the default configuration settings, doing so requires more effort on our part to denote and maintain the metadata describing the exceptions to the default configuration.

Use Portable Inheritance Mapping Strategies

Even if your JPA provider does implement the optional “table per concrete class” inheritance mapping strategy, it is best to avoid this if you need JPA provider portability.

It is also best to use a single inheritance mapping strategy within a given Java entity class hierarchy, because support for

mixing multiple mapping inheritance strategies within a single class hierarchy is not required of JPA implementations.

Leverage the Latest Tools

Major integrated development environments (IDEs) now bundle several JPA-related tools. JDeveloper offers a wizard that can easily create JPA-based entity classes with appropriate annotations directly from specified database tables. With just a couple more clicks, the JDeveloper user can similarly create a stateless session bean to act as a facade for these newly created entity beans. NetBeans 6.0 offers similar JPA wizards, and the Eclipse Dali project supports JPA tools for the Eclipse IDE.

Add Spring to Your JPA

A developer can use Spring to write JPA-based applications that can be easily run in standard Java environments, web containers, and full application server EJB containers with no changes necessary to the source code. This is accomplished via the Spring container's ability to inject datasources configured outside of the code and to support transactions via aspect-oriented programming also configured outside of the code. The Spring framework enables JPA developers to isolate specifics of handling JPA in the various environments (Java SE standalone, Java EE web containers, and Java EE EJB containers) in external configuration files, leaving transparent JPA-based code.

Another feature Spring 2.0 offers JPA developers is the @Repository annotation, which is helpful in assessing

database-specific issues underlying a JPA PersistenceException.

Finally, the Spring framework provides a convenient mechanism for referencing some of the JPA provider extensions that are common across the JPA providers.

Methodology & Processes

Think out of Box

It may be possible that are working on a scenario/business problem relating to an industry vertical that you know well and you may feel the urge to provide a solution taking into account your prior knowledge. Resist this temptation to go with preconceived notations. In fact, architects who adopt this approach tend to either fail.

Understand the problem statement and the scope thoroughly and don't let your pre-existing preferences and concepts influence your decision.

Meticulously select Frameworks and Libraries

Understand the advantages and disadvantages of the alternative technologies in each layer (JSF vs. JSP with JSTL etc.) and when one is more suitable than the other.

It's important to have knowledge outside of the scope of the business problem. For example you should know the aspects pertaining to security e.g Man in the Middle, Denial of Service, Spoofing, encryption mechanisms etc.

Don’t push the cool technologies (JSF, EJB, JPA etc) everywhere and don’t do or avoid doing something because in general it is considered to be a good or bad practice. One has to decide based on the capabilities of existing framework and a rational consideration of the pros and cons. For example even using JSP with JSTL tags to do database queries may be the most suitable solution in some special case.

Don't reinvent the Wheel

Use common, proven frameworks like Apache Struts, JavaServer Faces, and Eclipse RCP. Use proven patterns.

Back when we first started helping educate our clients in how to use the then-emerging Java EE standards, we discovered that developing a framework for user-interface development significantly improved developer productivity over building UI applications directly to the base servlet and JSP specifications. As a result, many companies developed their own UI frameworks that simplified the task of interface development.

As open-source frameworks like Apache Struts began development, the switchover to these new frameworks would be automatic and quick. The benefits of having an open-source community supporting the framework would be readily apparent to developers, and that they would gain universal acceptance very rapidly -- not only for new development, but in retro-fitted applications as well.

What has proven surprising is that this turned out otherwise. We still see many companies maintaining or even developing new user-interface frameworks that are functionally

equivalent to Struts or JSF. There are many reasons why this could be true: organizational inertia, "not invented here" syndrome, lack of perceived benefit in changing working code, or possibly even a slight sense of hubris in thinking that you could do things "better" than the open-source developers did in a particular framework.

However, the time is long-past when any of these reasons is worth using as an excuse not to adopt a standard framework. Struts and JSF are not only well accepted in the Java community, but fully supported within the vendors like Oracle and IBM. Likewise, in the rich client arena, the Eclipse RCP (Rich Client Platform) has also gained wide acceptance for building standalone rich clients. While not a part of the Java EE standard, these frameworks are now a part of the Java EE community, and should be accepted as such.

Develop as per customer specifications

Know the specifications by heart and deviate from them only after careful consideration. Just because you can do something doesn't mean you should.

It is very easy to cause yourself grief by trying to play around at the edges of what Java EE anables you to do. We find developers dig themselves into a hole by trying something that they think will work "a little better" than what Java EE allows, only to find that it causes serious problems in performance, or in migration (from vendor to vendor, or more commonly from version to version) later.

There are several places in which not taking the most straightforward approach can definitely cause problems. A

common one today is where developers take over Java EE security through the use of JAAS modules rather than relying on built-in spec compliant application server mechanisms for authentication and authorization. Be very wary of going beyond the authentication mechanisms provided by the Java EE specification. This can be a major source of security holes and vendor compatibility problems. Likewise, rely on the authorization mechanisms provided by the servlet and EJB specs, and where you need to go beyond them, make sure you use the spec's APIs (such as getCallerPrincipal()) as the basis for your implementation. This way you will be able to leverage the vendor-provided strong security infrastructure and, where business needs require, support more complex authorization rules.

Other common problems include using persistence mechanisms that are not tied into the Java EE spec (making transaction management difficult), relying on inappropriate Java Standard Edition facilities (like threading or singletons) within your Java EE programs, and "rolling your own" solutions for program-to-program communication instead of staying within supported mechanisms like Java 2 Connectors, JMS, or Web services. Such design choices cause no end of difficulty when moving from one Java EE compliant server to another, or even when moving to new versions of the same server. Using elements outside of Java EE often causes subtle portability problems. The only time you should ever deviate from a spec is when there is a clear problem that cannot be addressed within the spec.

Finally, be careful about adopting new technologies too early. Overzealously adopting a technology before it has been integrated into the rest of the Java EE specification, or into a

vendor's product, is often a recipe for disaster. Support is critical and if your vendor doesn't directly support a particular technology, you should carefully consider if you should use it. People tend to focus too much on easing the development process and neglect to consider the long term consequences of depending on large amounts of code developed outside your organization which is not supported by a vendor. Many project teams are enamored with new technology (for example, the latest open source framework) and quickly become dependent upon it without considering the very real costs to the business. Frankly, the decision to use any technology beyond what you've bought from your vendors should be carefully reviewed by corporate architecture, business, and legal teams (or their equivalent in your environment); just as normal product purchasing decisions are evaluated. After all, with rare exceptions, most of us are in the business of solving business problems, not advancing technology for the sheer fun of it.

Plan for Java EE security proactively

Turn on Application Server security. Lock down all your EJBs and URLs to at least all authenticated users.

It's a continual source of astonishment to us how few customers we work with originally plan to turn on WebSphere Application Server's Java EE security. Only around 50% of the customers we see initially plan to use this feature. We have even worked with several major financial institutions that did not plan on turning security on; luckily this situation was usually addressed in review prior to deployment.

Not leveraging Java EE security is a dangerous game. Assuming your application requires security (almost all do); you are betting that your developers can better build a security infrastructure than the one you bought from the Java EE vendor. This is not a good bet. Securing a distributed application is extraordinarily difficult. For example, you need to control access to EJBs using a network-safe encrypted token. Most home-grown security infrastructures are not secure -- with significant weaknesses that leave production systems terribly vulnerable.

In particular, while products like Access Manager provide excellent security features, they alone cannot secure an entire Java EE application. They must work hand in hand with the Java EE application server to secure all aspects of the system.

Another common reason given for not using Java EE security is that the role-based model does not provide sufficiently granular access control to meet complex business rules. Though this is often true, this is no reason to avoid Java EE security. Instead, leverage the Java EE authentication model and Java EE roles in conjunction with your specific extended rules. If a complex business rule is needed to make a security decision, write the code to do it, basing the decision upon the readily available and trustable Java EE authentication information (the user's ID and roles).

Iterative and Incremental Development

Iterative development enables you to gradually master all the moving pieces of Java EE. Build small, vertical slices through your application rather than doing everything at once.

Java EE is big framework and if a development team is just starting with Java EE, it is far too difficult to try learning it all at once. There are simply too many concepts and APIs to master. The key to success in this environment is to take Java EE on in small, controlled steps.

This approach is best implemented through building small, vertical slices through your application. Once a team has built its confidence by building a simple domain model and back-end persistence mechanism (perhaps using JDBC), and has thoroughly tested that model, they can then move onto mastering front-end development with servlets and JSPs that use that domain model. If a development team finds a need for EJBs, they could likewise start with simple session facades atop container-managed persistence EJBs or JDBC-based DAOs (Data Access Objects) before moving onto more sophisticated constructs like message-driven beans and JMS. This approach is nothing new, but relatively few teams actually build their skills in this way.

Instead, most teams cave in to schedule pressures by trying to build everything at once -- they attack the View layer, the Model layer, and the Controller layer in MVC, simultaneously. Instead, consider adopting some of the new Agile development methods, such as Extreme Programming (XP), that foster this kind of incremental learning and development. There is a procedure often used in XP known as Model First that involves building the domain model first as a mechanism for organizing and implementing your user stories. Basically, you build the domain model as part of the first set of user stories you implement, and then build a UI on top of it as a result of implementing later user stories. This fits very well with letting a team learn technologies one at a time,

as opposed to sending them to a dozen simultaneous classes (or letting them read a dozen books), which can be overwhelming.

Also, iterative development of each application layer fosters the application of appropriate patterns and best practices. If you begin with the lower layers of your application and apply patterns like Data Access Objects and session facades, you should not end up with domain logic in your JSPs and other View objects.

Finally, when you develop in thin vertical slices, it makes it easier to start early in performance testing your application. Delaying performance testing until the end of an application development cycle is a sure recipe for disaster.

Leverage Java EE capabilities

Commit to building real Java EE applications that truly leverage Java EE function.

One of the most disturbing things we've seen more than once is an application that claims to "run in WebSphere" but isn't really a WebSphere application. We've seen several examples where there is a thin piece of code (perhaps a servlet) in WebSphere Application Server and all of the remaining application logic is actually in a separate process; for example, a daemon process written in Java, C, C++ or whatever -- but not using Java EE -- does the real work. This is not the recommended approach. Virtually all of the qualities of service that Application Server provides aren't available to such applications. This can be quite a rude awakening for folks that think this is a WebSphere

Application Server application. Ensure that your solution is delivered in a vendor-neutral format.

Summary

Clarity makes it easy for the stakeholders to quickly and concisely understand your proposed solution. As you develop and document your solution, ask these questions to keep your solution on track:

• Am I providing a solution to the business problem posed, or am I solving what I want to solve?

• Is my solution clearly documented and does it reflect the intent?

• Is my solution simple and concise, while clearly solving the business problem as presented to me in the scenario?

When working with Java EE, most of the services we need are provided by the application server. Therefore, the number of required dependencies is minimal. In most cases, Java EE provides configuration by exception, meaning there is very little configuration to be done, and sensible defaults are used in the vast majority of cases. When configuration is needed it is done through annotations, which allows me to get the whole picture just by looking at the source code, without having to navigate back and forth between XML configuration files and source code.

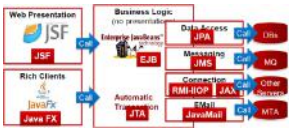

Figure 21: Java EE Framework 7

Java EE provides a platform for developing and deploying multitiered, distributed applications that are designed to be maintainable, scalable, and portable. Just as an office building requires a lot of hidden infrastructure of plumbing, electricity, and telecommunications, large-scale applications require a great deal of support infrastructure. This infrastructure includes database access, transaction support, and security. Java EE provides that infrastructure and allows you to focus on your applications.

Building distributed applications allows you to partition the software into layers of responsibility, or tiers. Distributed applications are commonly partitioned into three primary tiers: presentation, business rules, and data access. Partitioning applications into distinct tiers makes the software more maintainable and provides opportunities for scaling up applications as the demand on those applications increases.

Java EE architecture is based on the idea of building applications around multiple tiers of responsibility. The application developer creates components, which are hosted by the Java EE containers. Containers play a central theme in the Java EE architecture.

Servlets are one type of Java EE web component. They are Java classes that are hosted within, and invoked by the Java EE server by requests made to, a web server. These Servlets respond to those requests by dynamically generating HTML, which is then returned to the requesting client.

JSPs are very similar in concept to Servlets, but differ in that the Java code is embedded within an HTML document. The Java EE server then compiles that HTML document into a

Servlet, and that Servlet generates HTML in response to client requests.

JSF is a Java EE technology designed to create full and rich user interfaces. Standard user interface components are created on the server and connected to business logic components. Custom renderers take the components and create the actual user interface.

JDBC is a technology that enables an application to communicate with a data-storage. Most often that is a relational database that stores data in tables that are linked through logical relations between tables. JDBC provides a common interface that allows you to communicate with the database through a standard interface without needing to learn the syntax of a particular database.

EJBs are the centerpiece of Java EE and are the component model for building the business rules logic in a Java EE application. EJBs can be designed to maintain state during a conversation with a client or it can be stateless. They can also be designed to be short-lived and ephemeral, or can be persisted for later use. EJBs can also be designed to listen to message queues and respond to specific messages. Java EE is about a lot more than EJBs, although EJBs do play a prominent role.

The Java EE platform provides a number of services beyond the component hosting of Servlets, JSPs, and EJBs. Fundamental services include support for XML, web services, transactions, and security.

Extensive support for XML is a core component of Java EE. Support for both document-based and stream-based parsing of XML documents forms the foundation of XML support. Additional APIs provide XML registry service, remote procedure call invocation via XML, and XML-based messaging support.

Web services, which rely heavily on XML, provide support for describing, registering, finding, and invoking object services over the Web. Java EE provides support for publishing and accessing Java EE components as web services.

Transaction support is required in order to ensure data integrity for distributed database systems. This allows complex, multiple-step updates to databases to be treated as a single step with provisions to make the entire process committed upon success or completely undone by rolling back on a failure. Java EE provides intrinsic support for distributed database transactions.

Java EE provides configurable security to ensure that sensitive systems are afforded appropriate protection. Security is provided in the form of authentication and authorization.

Glossary

Sr #	Abbreviation	Description
1	RMI	Remote method Invocation
2	JNDI	Java Naming and Directory Interface
3	WAR	Web Application archive
4	JNI	Java Native Interface
5	JAAS	Java Authentication and Authorization Service
6	JCA	Java Cryptography Architecture
7	JAR	Java Archive
8	JTA	Java Transaction API
9	JAXR	Java API for XML Registries
10	JAX	Java API for XML
11	JAXM	Java API for XML Messaging
12	JCA	Java Connector Architecture
13	AJAX	Asynchronous JavaScript and XML
14	JMX	Java Management eXtension
15	BMP	Bean-Managed Persistence
16	BMT	Bean-Managed Transaction
17	JPA	Java Persistence API
18	JTA	Java Transaction API
19	JAX-RPC	Java API for XML-based Remote Procedure Calls
20	POJO	Plain Old Java Object

PART IV – Appendices

Appendix I: Approach and Methodology

RUP is based on software engineering best practices, offers a configurable framework and is scalable to support enterprise initiatives. Therefore, all aspects of RUP can also be applied to the development of an SOA. RUP provides a systematic approach to bridge the gap between business and IT to support a major area of concern, identification of services and how business processes are realized through execution of services. RUP also provides support for both the bottom-up and top-down approaches by acknowledgement of existing design elements and through activities such as architectural analysis to identify architectural elements such as services.

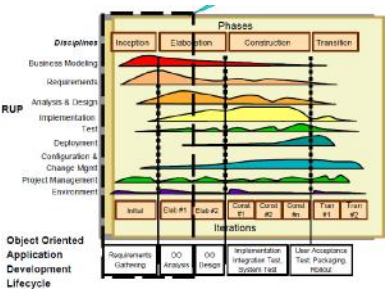

Figure 22: RUP Methodology

Approach – Solutioning

Domain decomposition represents a top-down approach where business domains are decomposed into functional areas across the value net. Through this technique we can establish the scope of the effort. After domains have been decomposed into functional areas, each area can then be further decomposed into processes and sub-processes and high-level

business use cases. Experience shows that the business use cases are considered good candidates for exposure.

Existing asset analysis represents the bottom-up approach where we analyze and leverage APIs, transactions and models from legacy and packaged applications as possible candidate services.

Goal-service modeling provides a middle-out approach that relates services to goals, sub goals, KPIs and metrics of the enterprise. This technique provides a certain level of validation in the form of a completeness check and may reveal candidate services that were not identified through the top-down and bottom-up activities.

Finally, subsystem analysis expands on sub-systems identified during domain decomposition and specifies interdependencies and flows between them. A key aspect of the identification step is that it employs a meet-in-the-middle approach including a combination of top-down, bottom-up and middle-out analysis techniques. In many cases a pure bottom-up approach is taken, however, this approach typically leads to poor definition of services that are driven mainly by architecture of legacy application interfaces and not from a business perspective.

Appendix II: Java EE Vs Dot Net

Java EE Or .Net - An Unbiased Opinion

Java EE

• Java EE has better ORM-support than .NET. JPA, Hibernate etc.NET comes with Entity Framework, but it isn't as lean as JPA. In .NET world there are some ports from Java available like e.g. NHibernate. .NET seems to more rely on stored procedures, whereby in Java everything in domain objects outside the DB.

• Build systems and CI tools in Java are more advanced. Hudson, Maven and ANT are a really nice and lean solution. On .Net side you can use NAnt or a proprietary MS tool.

• Community: Java community rocks - in .NET you have to rely on MSDN. However - this can be considered as advantage and disadvantage as well. You will find answers for your issues and queries.

• Java is the most popular language and is #1.

• Java IDEs: If you are thinking about .NET vs. Java EE you have already allocated some budget for tooling (otherwise the choice would be obvious). In that case you should look at IntelliJ first. It is the best, commercial IDE. If you are looking for free IDEs, start with NetBeans (best out-of-the-box experience) and Eclipse (the most popular one). Java tool ecosystem is remarkable.

• Integration: Java integration story is excellent. One gets connectors for SAP, AS400, files etc. for free. ESBs, LDAP servers are free.

• Framework / Libraries: If you need a specific library - you will find everything in Java. Before you are going to write a single line of code you should do some research first.

• Strategic consideration: with Java EE you are truly vendor-neutral. This is very handy in the price-negotiation phase. You can even go into production without any costs (subscriptions, licenses etc). Java EE apps are very portable. Java EE is not even dependent on Oracle - it was developed by all major vendors.

• Stability: Java language is stable for ages. There are no breaking changes. Java EE 5 and 6 platform is backward compatible as well. So you can deploy your legacy J2EE 1.X app to a Java EE environment and smooth-migrate it.

• Best practices: Java EE are about 10 years on the market. There are many applications and framework. The most mistakes are already made and not the best practices / patterns are very mature.

• In most of the .NET projects you will find open source extensions, as well as, commercial .NET tools. Such a mixture can be very "interesting" (regarding support issues)

• Operating System: you can choose whatever operating system you want to for production and development.

• Opensource: 80% of Java applications in production are built on open source tools. This is your ultimate insurance.

Dot Net

• Everything from one vendor.

• C# develops very fast and has nicer features, than Java. C# shines in the DSL area. LINQ is very nice for database (and tree-like) queries (however you should look at Scala and Groovy if you need similar features).

• You don't have to search for a given solution as the choice is clear. It can save you a considerable amount of time.

• Most of the tools at MSDN - there is no need to evaluate tools and IDEs in advance.

• Visual Studio .NET is a very good environment with very good DB and team extensions. With a commercial plugin (e.g. Resharper), it good, as IntelliJ.

• The UI best practices are more consistent, than in Java. ASP.NET MVC is e.g. the way to go in .NET. In Java EE you will have to choose between JSF, Wicket, GWT and Struts 2.

• The language integration in .NET is seamless. You can easily call methods from C# to VB.NET. In Java it is hard to call Scala functions from Groovy.

• SharePoint, Exchange and Office Products are very well integrated with .NET. You can access them with Java, but will need third-party libraries.

• Continuous Integration comes already with Visual Studio Team System.

Appendix III: Open Source Development

Separating the hype from reality isn't easy when it comes to Open Source. Not only has it become the technological buzzword, but it has also become the epicenter of a great deal of controversy: from copyright laws and intellectual property debates to freedom of speech and arguments about free-market competition. The J2EE market has evolved swiftly, first by going through a phase of consolidation and now by entering a phase of commoditization. This second phase has been driven largely by the fact that in order to show value, application server vendors can no longer rely on their core application server. This has created a market of value-added offerings, particularly in the area of development tools and development productivity. Many Open Source tools and frameworks showcased in this book are in this category.

Open Source is also changing the way programming is being studied in universities around the world; new generations of programmers leaving academia and entering the workplace have either used or contributed to Open Source. Students nowadays can learn by examining enterprise-level software that displays contributions from a great many sources from around the world.

At corporate IT departments worldwide, programmers are rallying behind Open Source projects like Ant, JUnit, Tomcat, and JBoss. Though the battle for the acceptance of Open Source has been largely fought at the level of the programmer and middle management, upper management, given the recent impact of Linux on corporations, is beginning to see the many advantages of Open Source, especially in the area of

enterprise Java. Organizations seeking to reduce software development expenses have found that Open Source software (OSS) provides a lower cost of ownership when compared to commercial offerings, primarily because Open Source software is free, both in price and restrictions.

Advantages and Disadvantages of Open Source

Many organizations are using Open Source projects to varying degrees in daily development. Some organizations use Open Source only during the development phases so that it doesn't affect any production environment. These organizations might use Open Source for building, unit testing, or integrated development environments (IDE). Organizations may also use Open Source libraries as a form of reuse for activities such as logging and XML parsing. Open Source application servers, web containers, and Common Object Request Broker Architecture (CORBA) servers can be used to provide the infrastructure. Organizations using Open Source are discovering there are some compelling reasons for using Open Source besides the financial benefits. Unfortunately, these organizations are also discovering that there are some disadvantages as well.

Advantages

The most obvious and compelling reason to use Open Source is the initial lower cost of ownership. Organizations are free to copy and distribute software to multiple developers and users. Consider an application with an installed base of 100 users and a 10-person development team using a $500 licensed commercial product. This would total $55,000 in expenses. With Open Source products the organization could

immediately eliminate the large expense and increase the install base without incurring additional expenses. Other financial benefits can be realized as well. Because Open Source is free to copy, the expense of license management isn't incurred. In addition, legal departments only have to review and approve an Open Source license once for all projects using that license rather than each time for each commercial product license. Using popular Open Source projects can reduce training expenses by providing a larger resource pool. Developers can be hired from outside the company with existing knowledge of Open Source frameworks. It's often difficult to hire developers that have knowledge of a proprietary commercial framework. Industry support is another reason to consider Open Source. Many major companies such as IBM, Sun, Oracle, BEA, and Borland are using Open Source projects. These organizations have a vested interest in the project's success because their products rely on the framework. Contributors to the Java Open Source projects aren't necessarily the independent programmers writing code in their spare time anymore.

Many of these large companies have departments dedicated to Open Source. In addition, many of the Open Source projects such as Eclipse, NetBeans, and Tomcat were initially donated by large corporate backers. Consider the use of Open Source as a means of expanding your development team to include some of the best resources from all around the world. Access to the source is an important advantage of Open Source. The source code is the only 100 percent–accurate documentation. JavaDocs, marketing material, architectural diagrams, and instructions often aren't kept up to date.

Open Source projects are more agile than commercial products in their evolution. Often Open Source projects have shorter release cycles than their commercial counterparts, if for no other reason than the fact that most projects provide nightly snapshots or direct access to the source code repository. In addition, organizations don't have to wait for a vendor's next release to get a bug fixed. Having source code provides a means for the organization to fix the bug itself. Organizations willing to contribute to Open Source projects can also have influence on the future direction of the project. Unlike proprietary development, Open Source has the advantage of being reviewed and tested by potentially hundreds or thousands of users. Unit and regression testing is an important part of software quality. Open Source projects such as the Jakarta Commons project requires that JUnit tests be available and passed before version releases. Having access to JUnit tests can reduce risks by providing means of testing new releases against the unit tests of the currently utilized release. The results of the tests can be used as a risk management tool to determine the impact of an upgrade on a project. Open Source can contribute to an individual's career development. Developers can use the source code to learn new techniques or APIs. Open Source can also lower the barriers of entry by allowing for more economical means of evaluating new technologies. For inexperienced Java developers, contributing to an Open Source project may be a way of demonstrating knowledge to an employer or potential employer. It's common for developers to evaluate a technology and prototype proof-of-concept applications using an Open Source project as a development environment while deploying on a commercial platform for production use.

Disadvantages

Open Source projects have been documented poorly. In addition, Open Source software usually doesn't have a recognized company behind it to provide support, whether it's free or paid for. These disadvantages are changing though. A new market has grown up around Open Source to provide quality documentation and support—for a price. In addition, many of the projects have active newsgroups or forums that can be effectively used to troubleshoot an Open Source application.

Open Source projects can also be plagued with backward-compatibility problems. Open Source projects don't take backward compatibility into consideration as much as commercial organizations. Yet, at the same time open source projects tend to be more daring when it comes to innovation and trying radically different ways to approach a problem.

The biggest disadvantage of Open Source is lack of marketing dollars. Often, organizations aren't aware of the existence of an Open Source project or how it might apply. Open Source projects don't have conference booths, magazine advertisements, or salespeople explaining the problems they can solve. Open Source projects also depend on the enthusiasm and number of collaborators as well as the areas that their efforts are focused on. For example, lack of documentation and administrative tools is a common complaint with regards to Open Source projects.

This brings to mind the successful emergence of Open Source projects with heavy commercial backers such as the Eclipse project, which is backed by IBM and thus has very good

documentation given the resources available, namely, IBM staff technical writers and editors.

Appendix IV: Sizing and Capacity Planning

Benchmark your code on hardware similar to what you'll be using in production, identify any bottlenecks, then determine how much of a workload your current hardware can handle, and/or how much hardware horsepower you need to handle your target workload.

A brief overview of the process:

• Create User Scenarios

• Add monitoring capability

• Add User Load

• Analyze results

• Remediate

• Rinse & Repeat

Step I: Create User Scenarios

Set up an environment to test against. This should be a fairly close your production hardware if possible, otherwise you will be left extrapolating your data. Set up your servers, accounts, websites, bandwidth, etc. Even if you do this in VMs that's OK just as long as you're prepared to scale your results.

Step II: Add Monitoring Capability

You'll need metrics to monitor e.g how many requests get through to the web servers, and how many requests can squeeze through per second before users start getting a response time of over two seconds. You will have to monitor RAM, CPU and disk usage to make sure that the load balancer can handle the connections. might need to review web server log files, start performance counters, or rely on the reporting ability of your stress test tool. Aspects required to be monitored:

• CPU usage

• RAM usage

• Disk usage

• Disk latency

• Network utilization

You might also choose to look at SQL deadlocks, seek times, etc depending on what you're specifically testing.

Step III: Add User Load

Simulate a test load. There are plenty of tools that can do this, with configurable options:

• JMeter (Web)

• Apache Benchmark (Web)

• Grinder (Web)

• httperf (Web)

• WCAT (Web)

• Visual Studio Load Test (Web)

• SQLIO (SQL Server)

Choose a number and let's say you're going to see how the system responds with 10,000 hits a minute. It doesn't matter what number you choose because you're going to repeat this step many times, adjusting that number up or down to see how the system responds.

Ideally, you should distribute these 10,000 requests over multiple load testing clients/nodes so that a single client does not become a bottleneck of requests. For example, JMeter's Remote Testing provides a central interface from which to launch several clients from a controlling JMeter machine. Press the magic Go button and watch your web servers melt down and crash.

Step IV: Analyze Results

So, now you need to go back to your metrics you collected in step 2. You see that with 10,000 concurrent connections, your high availability proxy box is barely breaking a sweat, but the response time with two web servers is a touch over five seconds. Remember, your response time is aiming for two seconds. So, we need to make refinements to the configurations.

Step V: Remediate

Now, you need to speed up your website by more than twice. So you know that you need to either scale up, or scale out. To scale up, get bigger web servers, more RAM, faster disks. To scale out, get more servers.

Use your metrics from step 2, and testing, to make this decision. For example, if you saw that the disk latency was massive during the testing, you know you need to scale up and get faster hard drives. If you saw that the processor was sitting at 100% during the test, perhaps you need to scale out to add additional web servers to reduce the pressure on the existing servers. There's no generic right or wrong answer, there's only what's right for you. Try scaling up, and if that doesn't work, scale out instead. Or not, it's up to you and some thinking outside the box. Let's say we're going to scale out. So I decide to clone my two web servers (they're VMs) and now I have four web servers.

Step VI: Rinse & Repeat

Start again from Step 3. If you find that things aren't going as you expected (for example, we doubled the web servers, but the response times are still more than two seconds), then look into other bottlenecks. For example, you doubled the web servers, but still have a crappy database server. Or you cloned more VMs, but because they're on the same physical host, you only achieved higher contention for the server's resources. You can then use this procedure to test other parts of the system. Instead of hitting the load balancer, try hitting the web server directly, or the SQL server using an SQL benchmarking tool.

The factors that affect the scalability of the server and long term strategy and tabulated below.

Determinant	Current Factor	Increase in 3 Yrs.
Number of Transaction	X	30%
Per cost transaction	Y	10%
Number of concurrent user	z	10%

Figure 23: Growth Projections

Capacity planning starts with measurement, in this case response time versus load. Once you know the degree to which the programs slows down with load, which is NOT a linear function, you can select a response time target, and then discover what resources it will take to meet that target for a given amount of load.

Performance measurement is always done with time units, as they are what users care about and they can be scaled up and down. Things like %CPU and IOPS are system-specific, so you only use them when you have planned the system and measured it in pre-production, to act as a "surrogate" for the thing you care about, time.

References

[1] UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design (2nd Edition)

[2] Architecting Enterprise Solutions: Patterns for High-Capability Internet-based Systems

[3] https://glassfish.java.net/

[3] http://arjan-tijms.omnifaces.org/2014/05/implementation-components-used-by.html

[4] http://java.dzone.com/articles/walking-through-java-ee-6

[5] http://tomcat.apache.org/tomcat-7.0-doc/cluster-howto.html

End