

Building Enterprise Application with J2EE

ABSTRACT

This material is intended for those who have already had some exposure to J2EE¹ technologies such as EJB and JSP/servlets, although architects and software engineers of all skill levels will find the design considerations, implementation techniques, and reusable code useful. Technically astute managers and other information technology professionals will also find it useful. This material will be of interest to any Java technologist building business applications using J2EE¹ from scratch and what is the kind of ROI using the same.

Table of Contents

Introduction	2
Guiding Principles to Build a Enterprise Application :	4
Applying Proven Design Patterns.....	5
Automating Common Functions	7
Practicality: Performance and Scalability.....	9
FAQ's	10
References	11
About The Author :	11

Introduction

Java 2 Enterprise Edition (J2EE¹) technology is becoming a pervasive platform for the development of Internet-based, transactional business applications. It provides a robust development platform upon which to build flexible, reusable components and applications. It is a powerful standard that is well-suited for Internet-based applications because it provides many of the underlying services such as HTTP request processing (Java servlet API), transaction management (Enterprise JavaBeans), and messaging (Java Message Service), just to name a few. However, J2EE¹ is also a complex and changing standard that leaves the technologist with many design decisions and performance considerations. Each component service adds a level of overhead to the application processing that must be considered. Additionally, there are a number of common business logic functions, such as error handling, that must be designed and developed for each component and application.

The J2EE¹ platform is designed to provide server-side and client-side support for developing distributed, multitier applications. Such applications are typically configured as a client tier to provide the user interface, one or more middle-tier modules that provide client services and business logic for an application, and back-end enterprise information systems providing data management. Figure 1.0 illustrates the various components and services that make up a typical J2EE¹ environment.

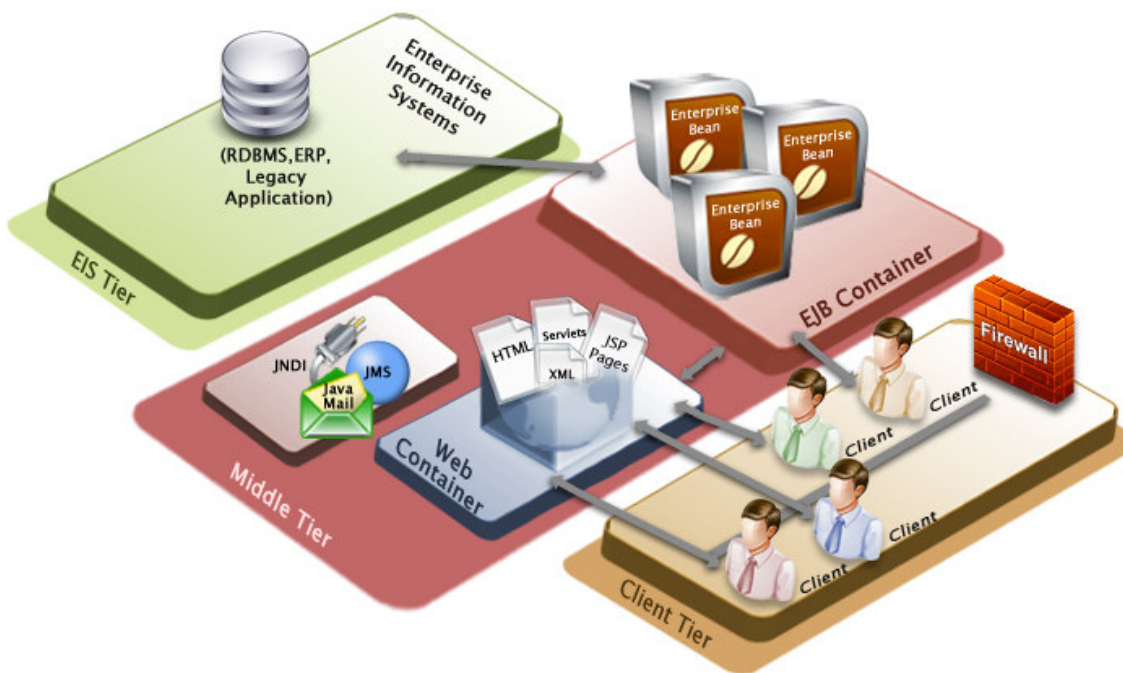


Figure 1.0 J2EE Environment

J2EE¹ Promises :

- Flexibility & Reliability
- Bottom line Productivity
- Performance
- Scalability
- Manageability
- Cost Effective
- Security
- Component based model
- Core runtime services
- Standardized approaches to integration
- Easier middleware development
- Wide portability via the JVM
- Complete Web services support
- Faster solutions delivery time to market
- APIs used are well documented

As illustrated, the J2EE¹ platform provides a multitier distributed application model. This means that the various parts of an application can run on different devices. The J2EE¹ architecture defines a *client tier*, a *middle tier* (consisting of one or more subtiers), and a *back-end tier*. The client tier supports a variety of client types, both outside and inside of corporate firewalls. The middle tier supports client services through Web containers in the *Web tier* and supports business logic component services through Enterprise JavaBeans (EJB) containers in the *EJB tier*. On the back end, the enterprise information systems in the *EIS tier* are accessible by way of standard APIs.

Guiding Principles to Build a Enterprise Application :

The goal of constructing the architecture is to create a development environment that can be used to build applications faster and with better performance, quality, and reusability. The following set of guiding principles are used to accomplish these goals:

- Applying proven design patterns to J2EE¹
- Automating common functions
- Using metadata-driven, configurable foundation components
- Considering performance and scalability

These principles are essential in driving the architecture and building the foundation for development. Much of software development in general and J2EE¹ development in particular can be optimized and automated through these concepts and their realization in the form of common foundation logic. Solid analysis of design choices as input to the architecture and application components is essential in order to provide solutions that balance the needs of rapid development, faster performance, higher quality, and greater reusability.

Figure I.1 shows the inputs and outputs of the architecture. This diagram essentially represents the guiding principles and the benefits that can be derived from applying them to application development. These principles provide the motivation and the basis for the approach to this study of developing applications using J2EE¹. Each aspect of the enterprise architecture within J2EE¹ will be studied for its behavior and characteristics. By using this information and applying the development principles and best practices, you can create an approach to effectively use the technology to reach our application development goals. The goals at the right side of Figure I.1, such as flexibility and reusability, should be considered and addressed from the beginning of any software development project. These types of goals are realized at two different levels: the software architecture level described earlier and the application component design. The reference architecture will guide much of the application design, so it is important to understand and distinguish these levels before undertaking enterprise software development. Each of the two levels will provide different types of benefits to both the end users and the development organization.

- J2EE11 API:
- Java Database Connectivity (JDBC)
- Remote Method Invocation (RMI)
- Java IDL
- Enterprise Java Beans
- Servlets and Java Server Pages (JSP)
- Java Message Service (JMS)
- Java Transaction API (JTA)
- JavaMail
- Java API for XML Processing (JAXP)
- Java Naming and Directory Interface (JNDI)

Applying Proven Design Patterns

A design pattern is a defined interaction of objects to solve a recurring problem in software development. There are a number of documented design patterns that represent proven solutions that you can use to solve common problems in object-oriented (OO) development. You can also apply many of these patterns to the J2EE architecture. One of the example of design patterns could be MVC using struts³ / spring² / hibernate⁴.

The cornerstone of the User Interaction Architecture is a generic implementation of the MVC architecture applied to J2EE. Jakarta Struts provides an excellent implementation that is readily available for developers to use. It provides a powerful tag library that integrates well with the Struts controller architecture to rapidly build transactional Web pages. The typical functionality provided by the controller architecture can be broken down into eight core responsibilities that revolve around four key abstractions: the user event, action, service, and Web page. The responsibilities center on the user-event abstraction that can be configured through metadata to drive the rest of the processing. The last step the controller servlet takes on each request is to forward to the next page, typically implemented as a JSP component. JSP components use should tag libraries as much as possible to encapsulate presentation logic and avoid large amounts of Java code interspersed with HTML. AJSP template mechanism provides a powerful way to apply a common look and feel to your application’s Web pages.

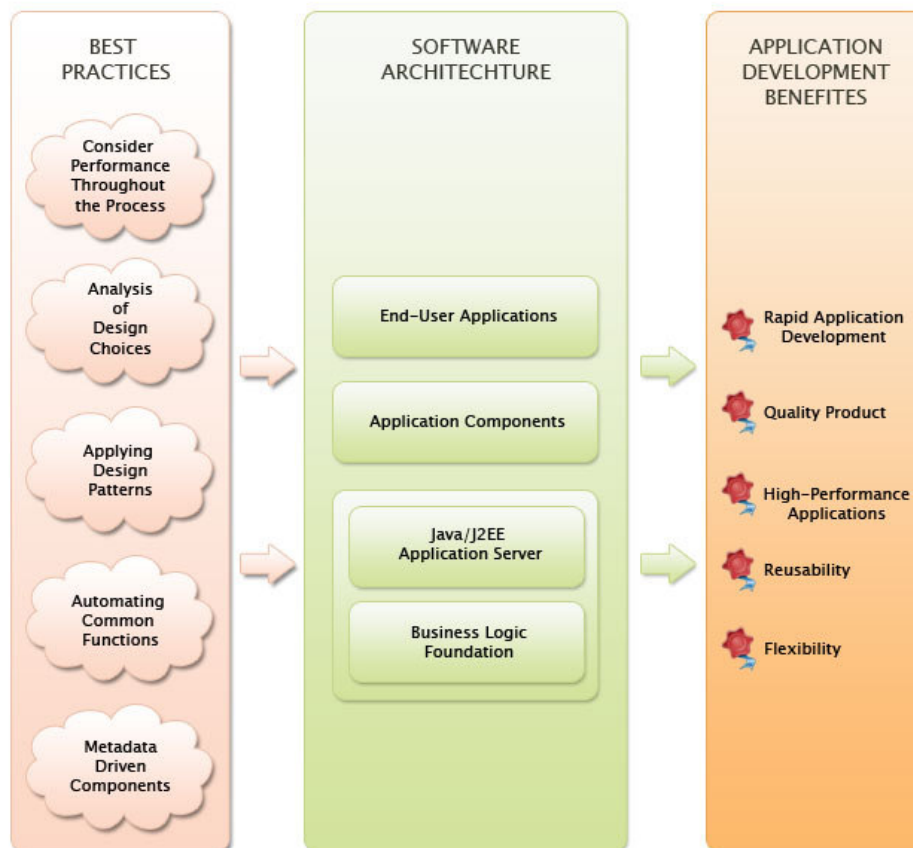


Fig 1.1 Reference Architecture and Principles

Automating Common Functions

The approach of automating common functions provides a number of benefits:

- Time is not wasted on monotonous, error-prone tasks.
- A higher-quality product through better-tested software; there is less total code to run through and it gets hit on every request; in essence, the foundation of much of the processing becomes a black box process with inputs.
- Automated functions and their common interfaces make it easier to develop and maintain consistent software across the application.

A set of configurable foundation components that automate basic elements of an application is often referred to as a framework. Building upon an earlier principle, many of these foundation components will be implemented using proven object-oriented design patterns. These framework components and patterns are what make up the reference architecture that will be used to rapidly develop quality J2EE¹ applications. As many developers know, there is a gap between the total sum of services needed to develop just purely application-specific logic and those that are currently provided with the development platform and this is where we need to think of all the common services that could be reused in our application .

The automation capabilities within technical frameworks provide a high level of reusability across applications. Reusability is of course the “Holy Grail” of object oriented software development. However, it has been very hard to achieve in many practical settings. Given a strategic application architecture and the set of guiding principles, you can position yourself to benefit from software reuse. The Enterprise JavaBean specification goes a long way toward having standard, reusable business components across applications. However, it is the role of the application architecture on top of J2EE to enable those components to be reused. It is important to have an application architecture that easily allows components to be plugged in to the rest of the system without adding significant overhead.

J2EE Frameworks :

- Struts³
- XDoclet⁸
- Tapestry⁹
- JSF¹⁰
- Velocity⁷
- Spring²
- Seam¹¹
- Hibernate⁴
- IBATIS⁶

Use Metadata-Driven Components

Metadata is usually defined as data that describes other data. Some examples of this could be the list of properties and their respective data types for a given business object, or it could be the form name and associated configuration information for a Web page. Much of the metadata that defines these components comes from design models described in UML. The principle of using metadata to drive components again builds upon a previous principle, that of automating the tasks of software development. Metadata is used as an input to the “framework” services that automate and drive the behavior of J2EE components. This is applicable at all levels of the architecture. In the case of business objects, metadata can be used to define the business entities and their attributes. At the workflow or transaction level, metadata can be used to drive the process flow of complicated tasks. At the user interface level, it can define a particular Web page form and how it should be processed. All of these elements of applications can be abstracted and defined using metadata. The J2EE specifications themselves rely on different forms of metadata to configure and deploy components.

Not every process or function should be defined using metadata (everything in moderation, as they say). There are some drawbacks to this approach that should be considered and that may not make it the right approach for every task. A meta data driven abstraction usually will add some overhead to the execution of the task when compared to explicit lines of code used to do the same job. This overhead is typically negligible when compared to something like a single database I/O request. However, it should be considered nonetheless in the overall approach to software development, especially where transaction throughput is essential to the success of an application.

Another potential drawback of this approach is the fact that it can make reading and debugging code a bit more difficult. A separate file or repository that contains the metadata determines portions of the flow through the code. There are a number of arguments to counteract this point, some of which have been mentioned here already. The primary argument is that these foundation components, which are configurable through metadata, become highly tested components that become almost like a black box to the rest of the application. Once you have these components working correctly, very little time is spent looking at the “framework” code. The behavior of an application can be determined simply by looking at the client code and the metadata inputs to the service.

Practicality: Performance and Scalability

The last principle, essentially performance engineering, is one that underlies all else. Avoiding this topic until the final phases of any project can have serious consequences. The quickest thing (no pun intended!) that will keep people from using your system is poor performance, especially in today's fast-paced Internet world. Business application users are accustomed to the performance of client-server applications over private networks and consumers or Internet users are very impatient when it comes to waiting for a Web-site page to load. Thus, although it is true that computers are getting faster and more hardware is always an option (if you built a scalable solution), you must keep a watchful eye and build performance into the development process from the very beginning. It must be a part of the design process because it often involves trade-offs with other aspects of a system, most often the flexibility that an application provides to the user.

Java, the language itself, can quickly approach the performance of C/C++ in many situations, a language widely regarded as a high-performance choice for even the most demanding applications. This is primarily due to the evolution of just-in-time (JIT) compilers that now aggressively translate Java byte code and perform code optimizations. This is particularly true on the server side, where you typically have a large set of Java classes that will be executed many times. The initial overhead of performing the translation into native instructions is usually not worth mentioning, and thus in theory, the majority of the code should be comparable to compiled C++ code. One weakness that Java still has when compared to C++ is the garbage collection process, which adds some overhead. However, the programming benefits are well worth the minimal cost involved in terms of memory allocation and management, so this really does not even become an issue. In fact, as processor speeds continue to increase, the difference between the two languages themselves is likely to become almost insignificant. However, component services provided by J2EE¹ add another layer on top of the language, and you must look very closely at the impact that component services have on the application's overall performance. While J2EE¹ provides many valuable services, such as object persistence and naming and directory services, their benefits must be weighed against their costs.

Many solutions will involve using Enterprise Java services in cases in which they provide the most benefit, but not as a standard across the board. This is a common tendency of building J2EE¹ architectures, to use the enterprise components across the board from front-to-back in the software architecture. A scalable architecture is a must for almost any system, and design guidelines discussed here for each layer of the architecture must be applied when deciding on the foundation for software components as well as in building the individual components themselves.

FAQ's

Q. Price of Acquisition and licensing?

A. J2EE is distributed freely. Most of the frameworks being used with J2EE are open source and can be extendible like Struts , Hibernate , Spring which are widely use and are being supported by the different organizations like Apache etc. Though there are some tools and applications servers by IBM , Oracle and BEA which charge their clients for the support. So in all Java offers a wide variety in terms of Free and Paid tools.

Q. Industry Momentum?

A. J2EE is widely used across different industries for building enterprise level applications and one can easily find trained man power who can do wonders with J2EE.

Q. Dependency on vendor?

A. You are not bound with a particular vendor and generally all new versions launched with in J2EE are backward compatible and exisisting code can be reused if we want to upgrade to the new version. In J2EE, standards compliance means you can fire any vendor and replace them with a better, more responsive vendor. We can create JAR(Java Archive), WAR(Web Archive) and EAR(Enterprise Archive) files to distribute our application or libraries. We can create these files using maven or ant scripts. Generally these files consist of classes, associated metadata and resources.

Q. Is it faster to develop apps in J2EE?

A. I would not say yes to this particular question because there are few UI related stuff that can be built faster in other technologies but from a long term perspective things can be delivered in a pretty much same time with quality and stability of the application.

Q. Performance?

A. Yes. J2EE comes with lots of API's and frameworks, which can be used to improve the performance of the overall application. Things like caching you can do memory management can do compilation at the server start time.

References

1. J2EE : <http://java.sun.com/j2ee/overview.html>
2. Spring : <http://www.springsource.org/>
3. Struts : <http://struts.apache.org/>
4. Hibernate : www.hibernate.org
5. UML : www.rational.com/uml/
6. IBATIS : <http://www.mybatis.org>
7. VELOCITY: <http://velocity.apache.org/>
8. XDoclet : <http://xdoclet.sourceforge.net/xdoclet/using.html>
9. Tapestry : <http://tapestry.apache.org/>
10. JSF : <http://www.oracle.com/technetwork/articles/javase/javaserverfaces-135231.html>
11. SEAM : <http://seamframework.org/>

About The Author :

Deepak Shokeen is currently working as Manager Technology with Calance Corporation and is been hands on different technologies in Java, J2EE¹. He has a overall experience of around 8 years. He has been part of different organizations like Honeywell, Global Logic, and Sapient. He has been involved in the different stages of software development life cycle and specialized in designing, development and maintenance. He has been involved in estimation, planning, handling teams and doing comparative study.

For more information about Calance products and services, call calance Sales information center at (412)-455-5469. To access information on internet, go to www.calance.com

© 2010 Calance Corporation. All rights reserved.

This case study is for informational purpose only. CALANCE MAKES NO WARRANTIES, EXPRESS OR IMPLIED, IN THIS SUMMARY. Calance and the calance logo are either registered trademarks or trademarks of Calance Corporation in the united States and/or other countries. The names of actual companies and products mentioned here in may be the trademarks of their respective owners.