Developing Applications for the Java EE 6 Platform

Student Guide

FJ-310-EE6

D65269GC11
Edition 1.1
May 2010
D67384

ORACLE

This page intentionally left blank.

This page intentionally left blank.

Table of Contents

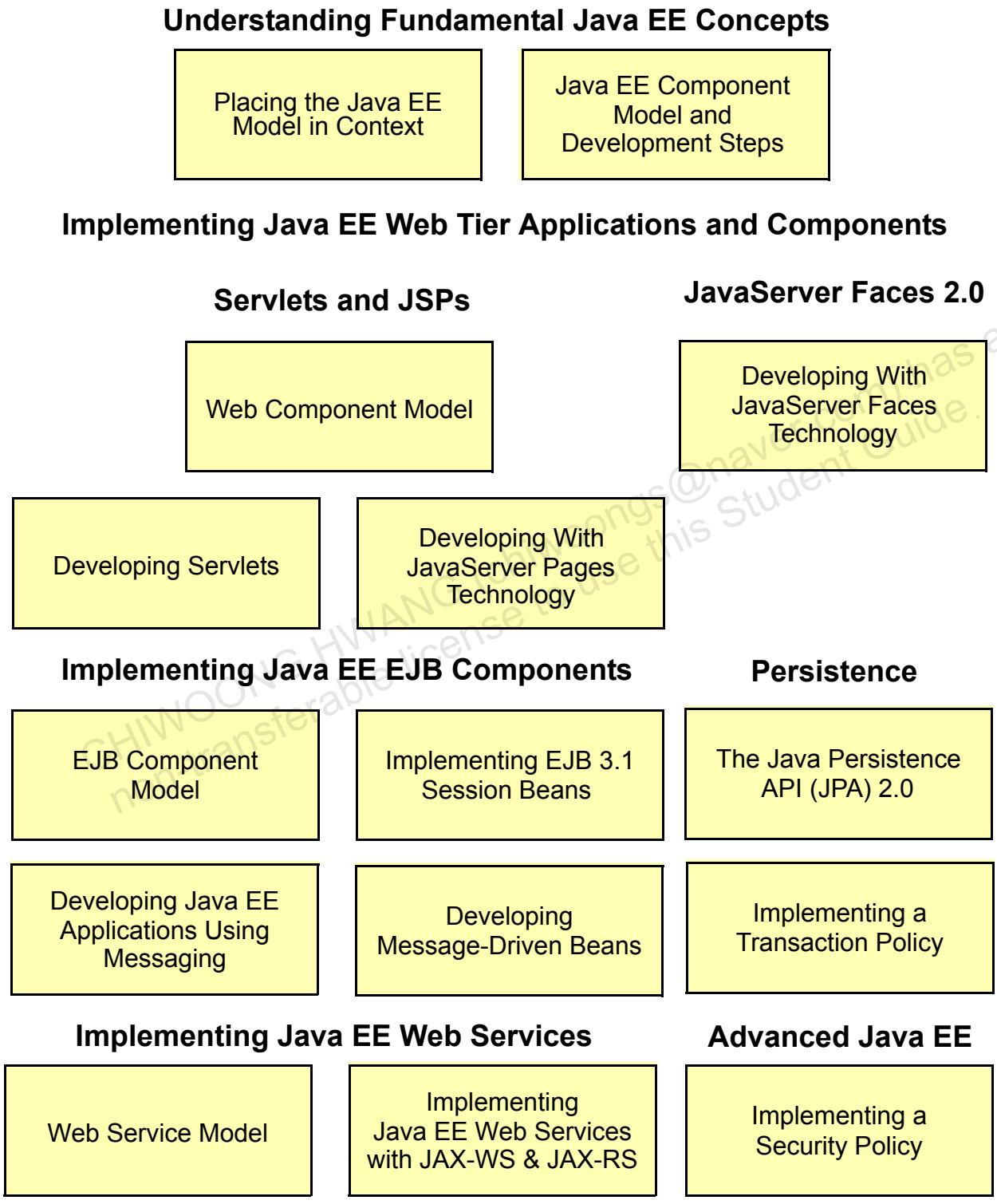
About This Course

Course Goals

Upon completion of this course, you should be able to:

- Describe the application model for the Java™ Platform, Enterprise Edition (Java™ EE platform) and the context for the model
- Develop a web-based user interface using Java Servlet and JavaServer Pages™ (JSP™) technology
- Develop a web-based user interface using JSF (JavaServer Faces) technology
- Develop an application based on Enterprise JavaBeans™ (EJB™) technology and the Java Persistence API (JPA)
- Implement a basic web service with the Java EE platform using JAX-WS and JAX-RS

Course Map



Topics Not Covered

This course does not cover the following topics. Many of these topics are covered in other courses offered by Sun Educational Services:

- Basic Java programming language concepts – Covered in course SL-275: *Java™ Programming Language*
- Object-oriented design and analysis concepts and Unified Modeling Language (UML) – Covered in course OO-226: *Object-Oriented Design and Analysis Using UML*
- Advanced web development, Servlets and JSPs – Covered in course SL-314: *Web Component Development on Servlet and JSP Technologies*
- Advanced JSF development – Covered in course SL-340: *Developing Web Applications using JSF Technologies*
- Advanced web services development, XML, SOAP, REST, JAXB – Covered in course DWS-4050: *Developing Web Services Using Java Technology* and DWS-4120: *Developing Secure Web Services*
- Advanced EJB component development, design, and implementation – Covered in course SL-355: *Business Component Development with EJB Technology*
- Advanced database programming with the Java Persistence API – Covered in course SL-370: *Building Database-Driven Applications With Java Persistence API*
- Patterns for the Java EE platform – Covered in course SL-500: *Java™ EE Patterns*

Refer to the Sun Educational Services catalog for specific information and registration.

How Prepared Are You?

To be sure you are prepared to take this course, can you answer yes to the following questions?

- Are you experienced with the Java programming language?
- Are you familiar with distributed programming (multi-tier architecture)?
- Are you familiar with relational database programming?
- Are you familiar with component technology?

Introductions

Now that you have been introduced to the course, introduce yourself to the other students and the instructor, addressing the following items:

- Name
- Company affiliation
- Title, function, and job responsibility
- Experience related to topics presented in this course
- Reasons for enrolling in this course
- Expectations for this course

How to Use Course Materials

To enable you to succeed in this course, these course materials contain a learning module that is composed of the following components:

- Goals – You should be able to accomplish the goals after finishing this course and meeting all of its objectives.
- Objectives – You should be able to accomplish the objectives after completing a portion of instructional content. Objectives support goals and can support other higher-level objectives.
- Lecture – The instructor presents information specific to the objective of the module. This information helps you learn the knowledge and skills necessary to succeed with the activities.
- Activities – The activities take on various forms, such as an exercise, self-check, discussion, and demonstration. Activities help you facilitate the mastery of an objective.
- Visual aids – The instructor might use several visual aids to convey a concept, such as a process, in a visual form. Visual aids commonly contain graphics, animation, and video.

Conventions

The following conventions are used in this course to represent various training elements and alternative learning resources.

Icons

Additional resources – Indicates other references that provide additional information on the topics described in the module.

Note – Indicates additional information that can help you, but is not crucial to your understanding of the concept being described. You should be able to understand the concept or complete the task without this information. Examples of notational information include keyword shortcuts and minor system adjustments.

Typographical Conventions

`Courier` is used for the names of commands, files, directories, programming code, and on-screen computer output. For example:

Java Archive (JAR) files are normally given names that end in `.jar`.

`Courier` is also used to indicate programming constructs, such as class names, methods, and keywords. For example:

For example, the `getName` and `setName` methods represent the `name` property.

`Courier bold` is used for characters and numbers that you type. For example:

To list the files in this directory, type:
`# ls`

`Courier bold` is also used for each line of programming code that is referenced in a textual description; for example:

```
1 import java.io.*;
2 import javax.servlet.*;
3 import javax.servlet.http.*;
```

Notice the `javax.servlet` interface is imported to allow access to its life-cycle methods (Line 2).

`Courier italics` is used for variables and command-line placeholders that are replaced with a real name or value. For example:

To delete a file, use the `rm` *`filename`* command.

`Courier italic bold` is used to represent variables whose values are to be entered by the student as part of an activity. For example:

Type **`chmod a+rwx`** ***`filename`*** to grant read, write, and execute rights for *`filename`* to world, group, and users.

Palatino italics is used for book titles, new words or terms, or words that you want to emphasize. For example:

Read Chapter 6 in the *User's Guide*.
Each service uses a *state engine* that acts like a protocol checker.

Additional Conventions

Java programming language examples use the following additional conventions:

- Method names are not followed with parentheses unless a formal or actual parameter list is shown; for example:

  "The `doIt` method..." refers to any method called `doIt`.

  "The `doIt()` method..." refers to a method called `doIt` that takes no arguments.

- Line breaks occur only where there are separations (commas), conjunctions (operators), or white space in the code. Broken code is indented four spaces under the starting code.

- If a command used in the Solaris™ Operating System is different from a command used in the Microsoft Windows platform, both commands are shown; for example:

  If working in the Solaris Operating System:

  ```
  $CD SERVER_ROOT/BIN
  ```

  If working in Microsoft Windows:

  ```
  C:\>CD SERVER_ROOT\BIN
  ```

Module 1

Placing the Java™ EE Model in Context

Objectives

Upon completion of this module, you should be able to:

- Describe the needs of enterprise applications and describe how the Java Platform, Enterprise Edition 6 (Java EE 6) technology addresses these needs
- Describe the Java EE 6 platform application programming interfaces (APIs) and supporting services
- Describe the Java EE platform tiers and architectures
- Describe how to simplify Java EE application development using architecture patterns

Additional Resources

Additional resources – The following references provide additional information on the topics described in this module:

- Alur, Deepak, John Crupi, and Dan Malks. *Core Java EE™ Patterns: Best Practices and Design Strategies 2nd Edition*. Sun Microsystems Press: Prentice Hall PTR, 2003.
- Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns*. Reading: Addison Wesley Longman, Inc., 1995.
- Stelting, Stephen, and Olav Maassen. *Applied Java Patterns*. Palo Alto: Sun Microsystems Press, 2002.
- Java Community Process Web site, [`http://jcp.org/en/home/index`], accessed 1 August 2009.
- Eric Jendrock, Debbie Carson, Ian Evans, Devika Gollapudi, Kim Haase, Chinmayee Srivathsa. “The Java EE 6 Tutorial,” [`http://java.sun.com/javaee/6/docs/tutorial/doc/`], accessed 1 August 2009.
- “JSR-000316 Java Platform, Enterprise Edition 6 Specification,” [`http://jcp.org/en/jsr/detail?id=316`], accessed 1 August 2009.
- “JSR-000316 Java EE 6 Web Profile Specification,” [`http://jcp.org/en/jsr/detail?id=316`], accessed 1 August 2009.
- “Java BluePrints, Enterprise BluePrints,” [`http://java.sun.com/blueprints/enterprise/`], accessed 1 August 2009.
- “NetBeans Documentation, Training & Support,” [`http://www.netbeans.org/kb/`], accessed 1 August 2009.

The Requirements of Enterprise Applications

The Java EE platform is an architecture for implementing enterprise-class applications using Java and Internet technology. A primary goal of Java EE technology is to simplify the development of enterprise-class applications through a vendor-neutral, component-based application model.

The Java EE and the Java Technology Platforms

Figure 1-1 shows how the Java EE platform fits with the other Java technology platforms.

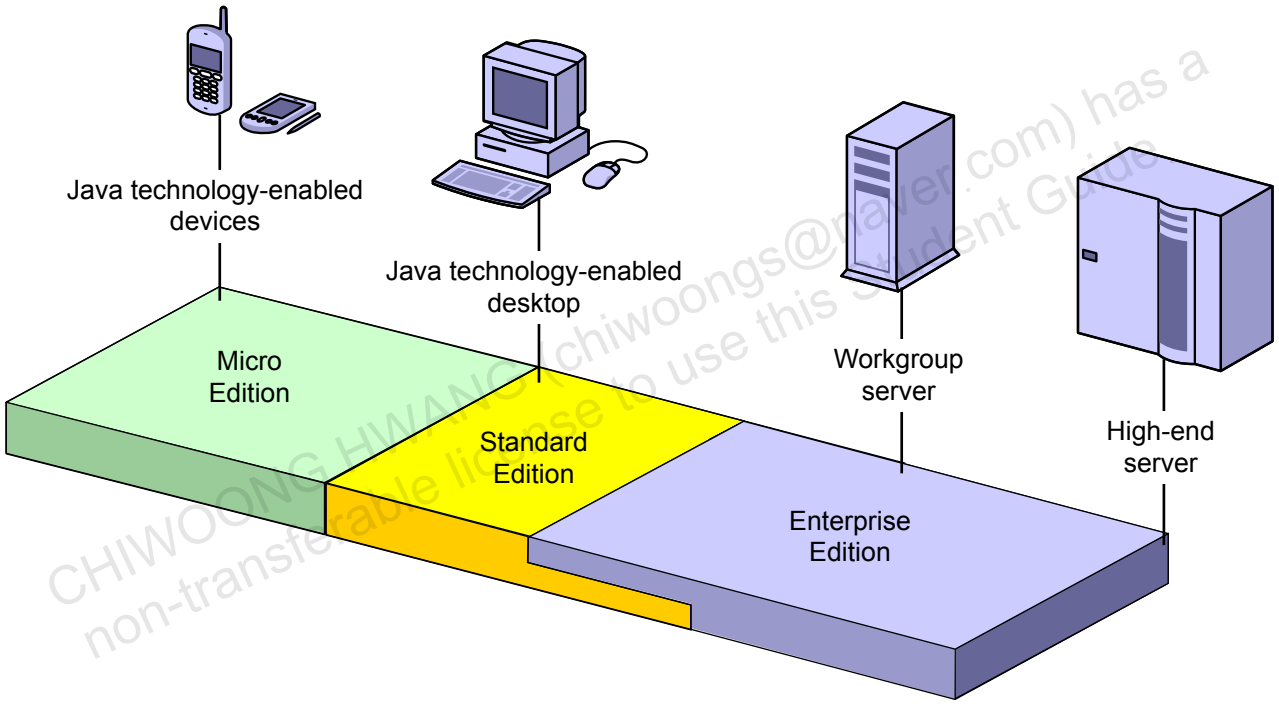


Figure 1-1 Java Technology Platforms

Java EE technology is used to develop distributed, enterprise-scale applications that, by their very nature, are often quite complex and resource intensive.

Enterprise Application Infrastructure Technologies

Figure 1-2 shows the enterprise infrastructure technologies that are required to expose the application logic and functionality that is provided in a single-user business application as an enterprise application.

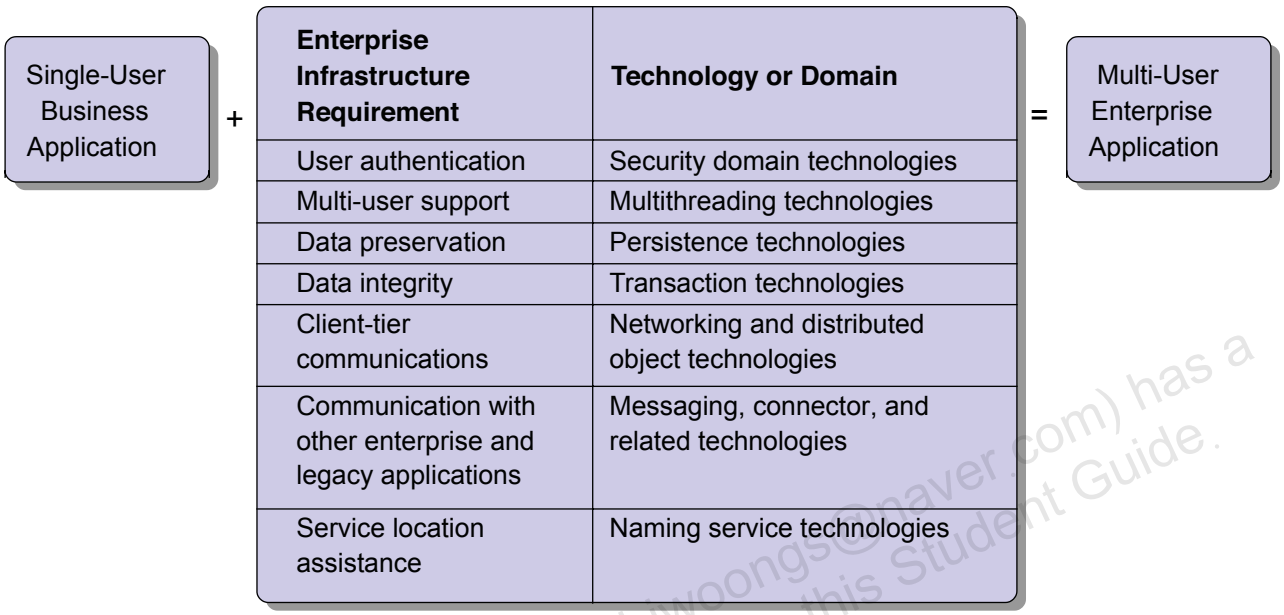


Figure 1-2 Enterprise Application Infrastructure Technologies

Developing enterprise scale applications is a difficult and time-consuming task that requires technical expertise, typically from a group of people who have separate roles and responsibilities, such as system architects, designers, and component developers.

When developing an enterprise application, a host of design considerations must be made, some of which involve trade-offs and compromises. The work performed by separate development teams must also be managed and coordinated. It is important to remember that the Java EE model exists to make a difficult job *tractable*, not to make it *easy*.

Java EE Technology Elements

At the heart of the Java EE platform is the Java Platform, Standard Edition (Java SE) technology. Figure 1-3 on page 1-6 illustrates how the Java EE specification is built on top of the functionality that is defined in the Java SE specification. The figure includes a number of acronyms that are defined here for your reference:

- Common Object Request Broker Architecture (CORBA)
- Enterprise JavaBeans™ (EJB™) components
- Java API for XML Web Services (JAX-WS)
- Java API for RESTful Web Services (JAX-RS)
- Java Management extensions (JMX™)
- Java Message Service (JMS) API
- Java Naming and Directory Interface™ (JNDI) API
- Java Transaction API (JTA)
- JavaServer Pages™ (JSP™)
- JavaServer Faces™ (JSF™)
- Remote Method Invocation (RMI)
- Structured Query Language (SQL)
- Java Persistence API (JPA)

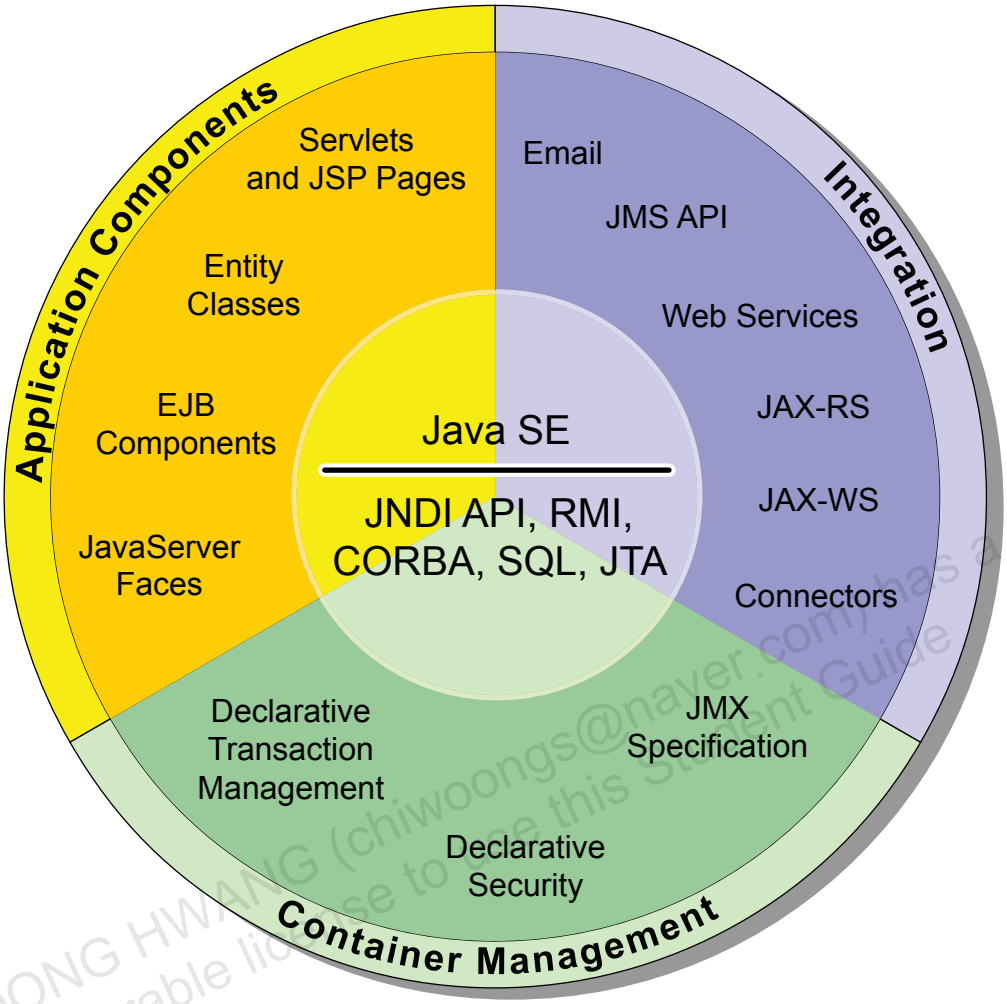


Figure 1-3 Java EE Technology Suite

The Java EE specification incorporates a suite of other technologies and specifications, in addition to those defined by Java SE, to provide a rich feature set and enhanced server-side functionality for enterprise application developers.

- Application components – Application developers use application components to create the application business logic and control elements.
- Integration – Integration elements allow a Java EE application to interact with and incorporate the functionality from other applications and systems, such as legacy systems or databases.
- Container management – Container management elements provide runtime support for Java EE application components.

Java EE and the Java Community Process℠

Figure 1-4 shows how a set of specifications that are maintained as part of the Java Community Process℠ (JCP℠) defines the roles and responsibilities of Java EE platform vendors, tools providers, and component developers.

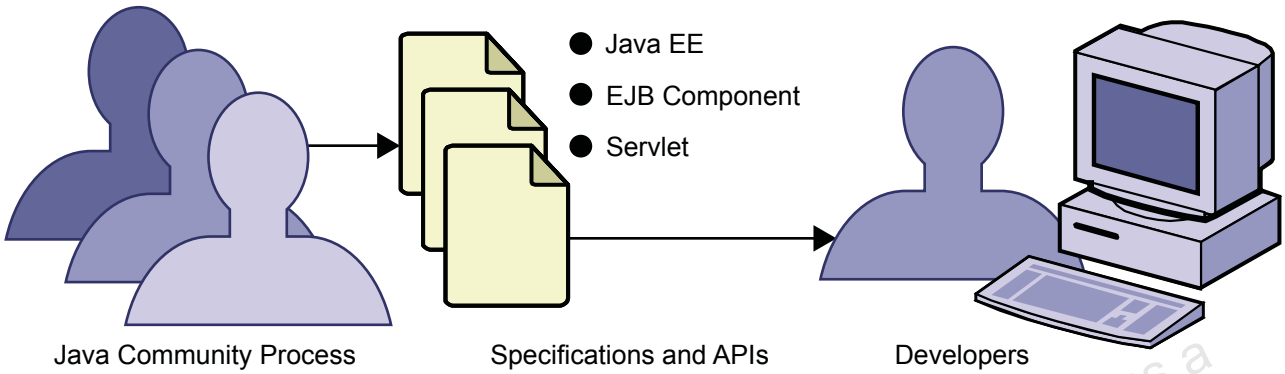


Figure 1-4 Java EE Specifications and the Java Community Process

The specifications outline the rules that each of these participants must follow when they develop Java EE technology components and supporting services. The standards-based approach helps to ensure that Java EE applications and application components are portable across a wide variety of deployment platforms. To find out more about the JCP, visit the JCP home page that is listed for your reference in “Additional Resources” on page 1-2.

The Java EE specification defines the types of components and the associated APIs that are available to Java EE application component developers. The Java EE specification also defines the infrastructure requirements for a robust, scalable, and reliable runtime environment for distributed, enterprise applications. Java EE server vendors use these specifications when they develop server elements.

Function of the Java EE APIs

The Java EE APIs define the contract between the application component developer and the platform provider using the interface mechanism that is defined by the Java programming language. As long as the application server implements the API set for the Java EE platform, the application component developer need not be concerned with how the vendor has chosen to implement the APIs. Figure 1-5 shows how the Java EE platform provides vendor-neutrality in the component layer using the APIs.

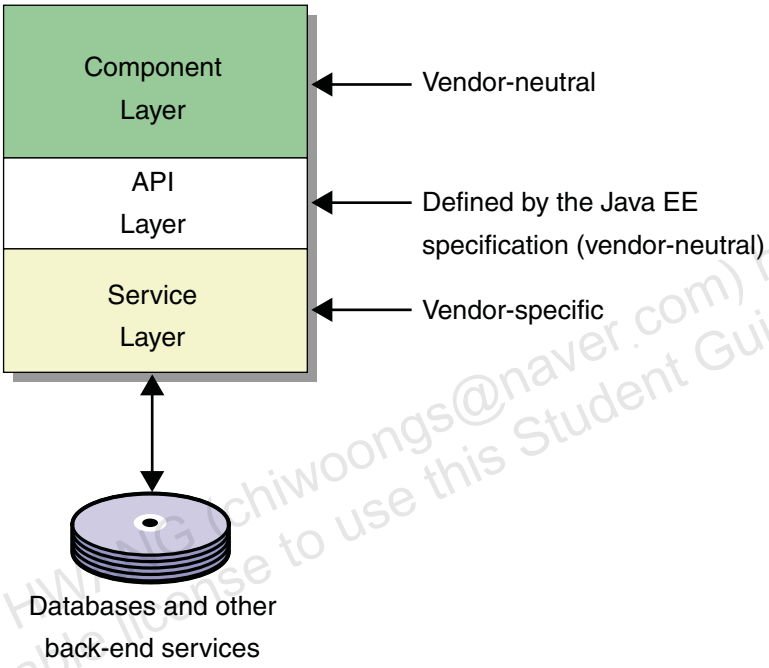


Figure 1-5 Component, API, and Service Layer

Java EE Containers

Java EE application components reside in containers from which they obtain runtime support. Figure 1-6 shows the four types of containers that are available in the Java EE platform, including the web container, the enterprise bean container, an embedded EJB container, and the application client container.

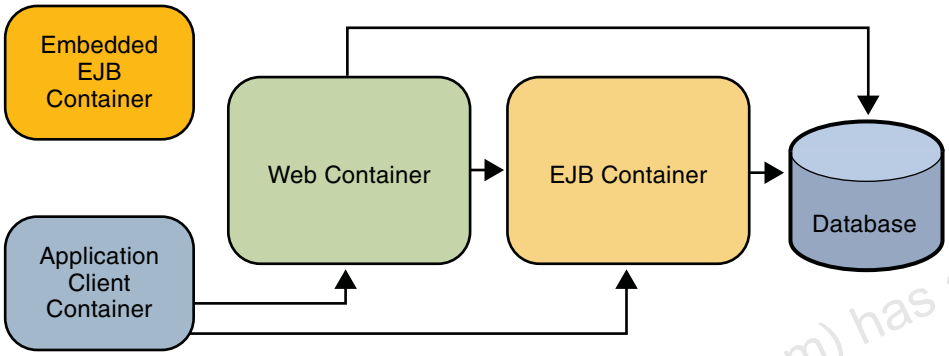


Figure 1-6 Java EE Component Containers

Each container type provides a support infrastructure that is tailored to the specific needs of the respective component types. Application components interact with other components and platform services using the protocols and methods that are provided by the container. All interactions with container-based components pass through the container. Consequently, the container can inject service and runtime support when necessary as part of this interaction.

Separation of Business Logic From Platform Services

A key feature of the Java EE platform is the strict separation of the application components from the general services and infrastructure. One of the main benefits of the Java EE platform for the application developer is that it makes it possible for developers to focus on the application business logic while leveraging the supporting services and platform infrastructure provided by the container and application server vendor.

For example, in an online banking application, the application component developers need to code the logic that underlies the transfer of funds from one account to another, but they do not need to be concerned about managing database concurrency or data integrity in the event of a failure. The application server infrastructure and services are responsible for these functions.

Figure 1-7 contrasts the tasks that are required of an application developer who builds an application and supporting services from the ground up to those of a developer who relies on an application component server for service-level and platform-level functions.

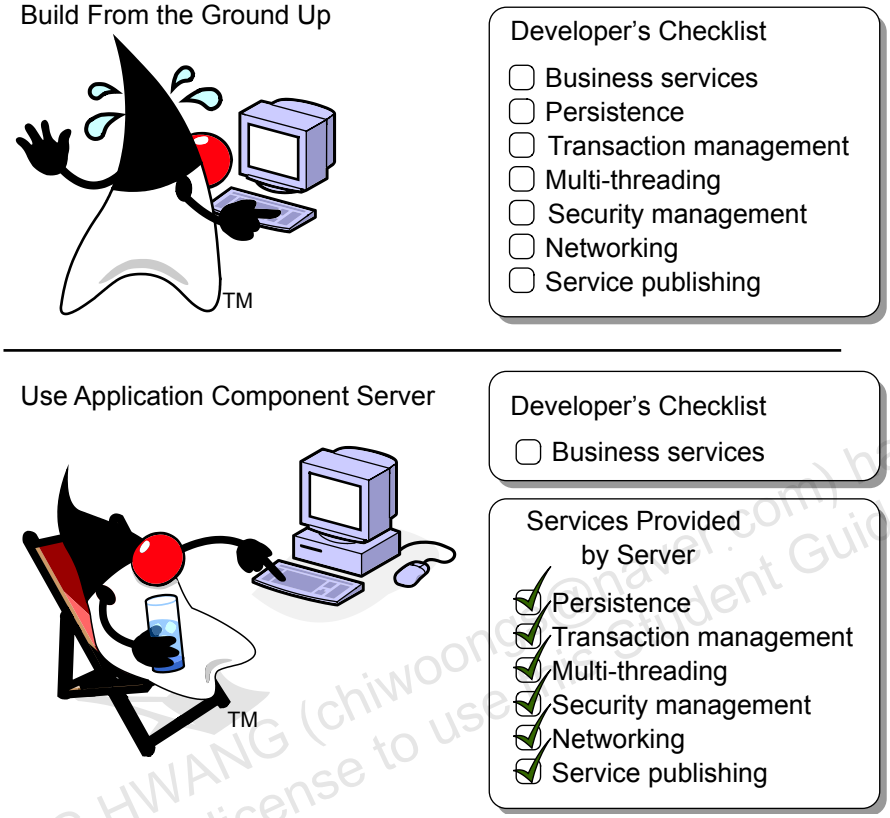


Figure 1-7 Advantages of Using Server-Provided Services

As you can see, relying on a component server for application support services dramatically reduces the amount of coding required of the application component developer.

Java EE Platform APIs and Services

The Java EE component model relies on both container-based and platform services for ancillary functionality that is not directly related to the application business logic. The Java EE platform goes beyond the traditional middleware server in terms of the range of services that it offers and the generality of application(s) that can be supported.

Java EE Service Categories

Java EE services can be grouped into the following categories.

- Deployment-based services – You request deployment-based services declaratively using XML in a file called a deployment descriptor or by using Java annotations. Deployment-based services might include:
  - Persistence
  - Transaction
  - Security
  - URL mapping
- API-based services – You request API-based services programatically. You must include code in the component to request these services. API-based services might include:
  - Naming
  - Messaging
  - Connector
- Inherent services – The container automatically supplies inherent services to components on an as-needed basis. Inherent services include:
  - Life-cycle
  - Dependency Injection
  - Threading
  - Remote object communication, such as RMI and CORBA
- Vendor-specific functionality – Vendor-specific functionality can include clustering, which addresses:
  - Scalability
  - Failover
  - Load balancing

Note – You should consider vendor-specific functionality, such as scalability and load balancing, more as a feature of the server than as a service.

Java EE API-Based Services

Figure 1-8 shows how the Java EE containers have access to a range of important API-based services as defined by the Java EE specification.

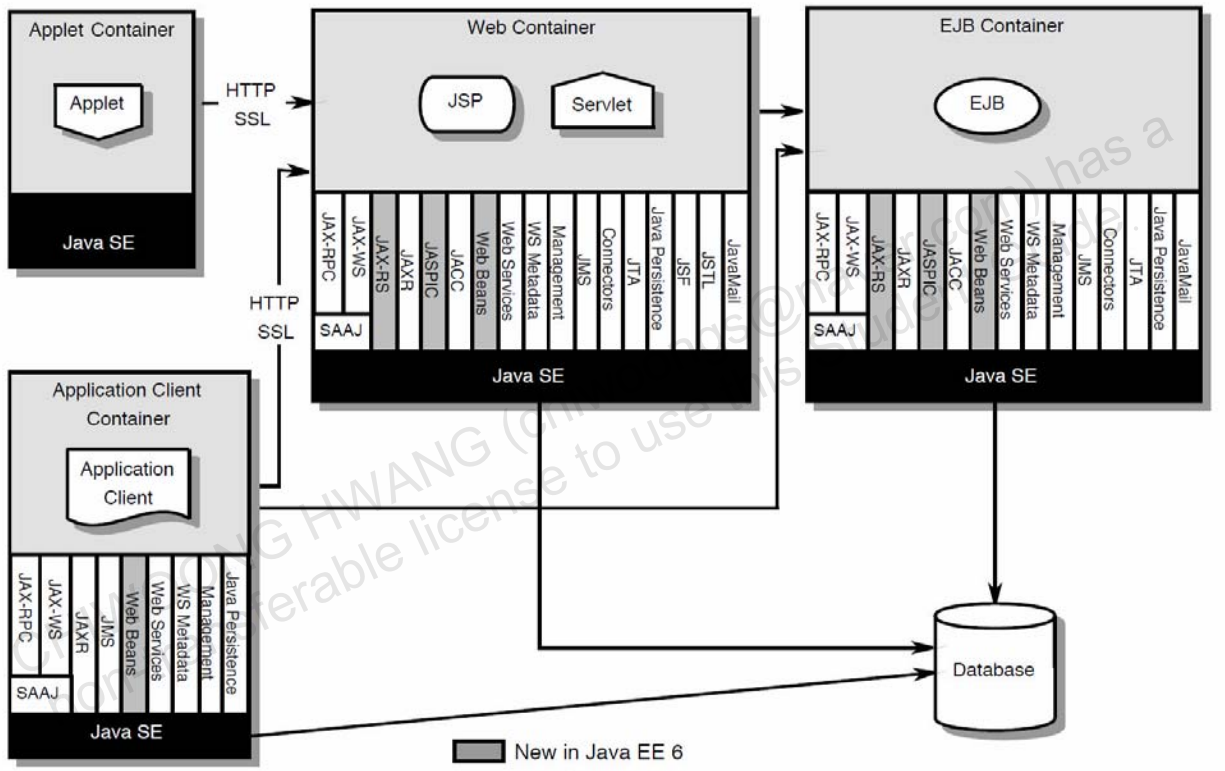


Figure 1-8 Java EE Service Infrastructure

The following list identifies the most important supporting services and APIs that are included in the Java EE 6 platform:

- Java DataBase Connectivity™ (JDBC™) API for database connectivity

  This API provides a vendor-neutral way for applications to complete relational database operations in the Java programming language, with SQL as the query medium.

- Java Naming Directory Interface (JNDI) API

  This API is used for vendor-neutral access to directory services, such as Network Information Service Plus (NIS+) and Lightweight Directory Access Protocol (LDAP). Java EE applications also make use of the JNDI API to locate components and services using a central lookup service.

- RMI over Internet Inter-Object Request Broker (ORB) Protocol (IIOP) and the Interface Definition Language (IDL) for the Java application

  Together, these services form a CORBA-compliant remote method invocation strategy. The strength of this strategy over Java RMI schemes is that it is programming-language-independent, so not all clients of a particular enterprise application need to be written in the Java programming language.

- JavaMail™ API and JavaBeans™ Activation Framework (JAF) API

  These APIs allow a Java EE software application to send email messages in a vendor-independent way.

- Java EE Connector Architecture (JCA) API

  This API allows the provision of integration modules, called *resource adapters*, for legacy systems in a way that is independent of the application server vendor.

- Java Message Service (JMS) API

  This is an API for sending and receiving asynchronous messages.

- Java Transaction (JTA) API

  This is an API by which software components can initiate and monitor distributed transactions. Java Transaction Service (JTS) specifies the implementation of a transaction manager, which supports the JTA 1.0 Specification at the high-level, and implements the Java programming language mapping of the Object Management Group (OMG) Object Transaction Service (OTS) 1.1 Specification at the low-level.

- Java Authentication and Authorization Service (JAAS)

  In the Java EE platform, JAAS might be used to integrate an application server with an external security infrastructure.

- Java API for XML Processing (JAXP)

  This API provides access to XML parsers. The parsers themselves might be vendor-specific, but as long as they implement the JAXP interfaces, vendor distinctions should be invisible to the application programmer.

- Web services integration features

  These services include, Simple Object Access Protocol (SOAP) for the Java application, SOAP with Attachments API for Java™ (SAAJ), Streaming API for XML (StAX), Java API for XML Registries (JAXR), JAX-WS, and JAX-RS. Together these services allow Java EE software applications to respond to SOAP and RESTful based web services requests and to initiate SOAP operations.

- Java Management Extensions (JMX) API

  This API exposes the internal operation of the application server and its components for control and monitoring vendor-neutral management tools.

- Timer services

  These services provide the ability to run scheduled background tasks.

- Java Persistence API (JPA)

  This API provides access to object relational mapping services enabling object oriented applications to persist object state in a database management system (DBMS).

Java EE Platform Tiers and Architectures

The Java EE specification outlines an architectural model that is based on tiers that developers are encouraged to use when they create a Java EE application. Tiering was historically motivated because of the division of labor that was associated with specialized servers, as well as by the formal definition of application responsibilities based on that division.

N-Tier Architecture

A Java EE enterprise application typically implements an N-tier architectural model. Figure 1-9 shows an example of the N-tier architecture model.

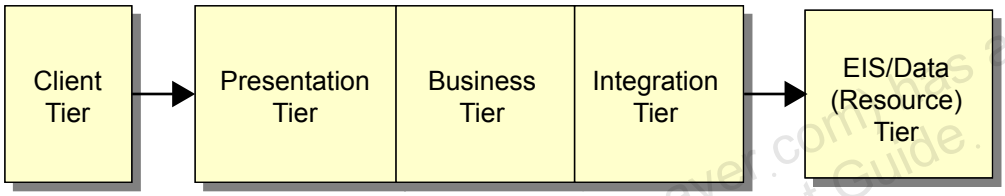


Figure 1-9 N-Tier Architecture Model

The N-tier architectural model describes any type of application architecture that programmatically separates application functionality across three or more tiers. The components in each tier, as well as the server-side infrastructure that is available in each tier, are generally uniquely suited to a particular task. For example, the presentation tier components and runtime infrastructure for an application that interacts with browser clients over the Internet must be able to process Hypertext Transport Protocol (HTTP) requests and generate responses that are formatted using Hyper-Text Markup Language (HTML). Similarly, the integration tier runtime infrastructure and associated components might support functionality for representing a back-end data store as an object model. The integration tier components synchronize changes in the data model across a set of data resources that are provided by the enterprise information system (EIS).

Programmatic interfaces typically define the boundaries between tiers. The interfaces are contracts that define the functionality that is available from a tier, as well as the data elements that are passed between the tiers. These interfaces must be well-designed and stable within the system. Changes to any tier have minimal impact on any other tier, as long as the interface definitions do not change.

The responsibility for developing and maintaining application components in each tier is often delegated to separate programming teams based on their area of expertise. For example, developers, who specialize in the creation of user interfaces, create presentation-tier components, while, developers, who have specific knowledge about a business domain, develop the related business-tier components.

Java EE Application Mapped to the N-Tier Model

As illustrated in Figure 1-10, you can typically map the functionality contained within a Java EE application to the middle tiers of an N-tier model.

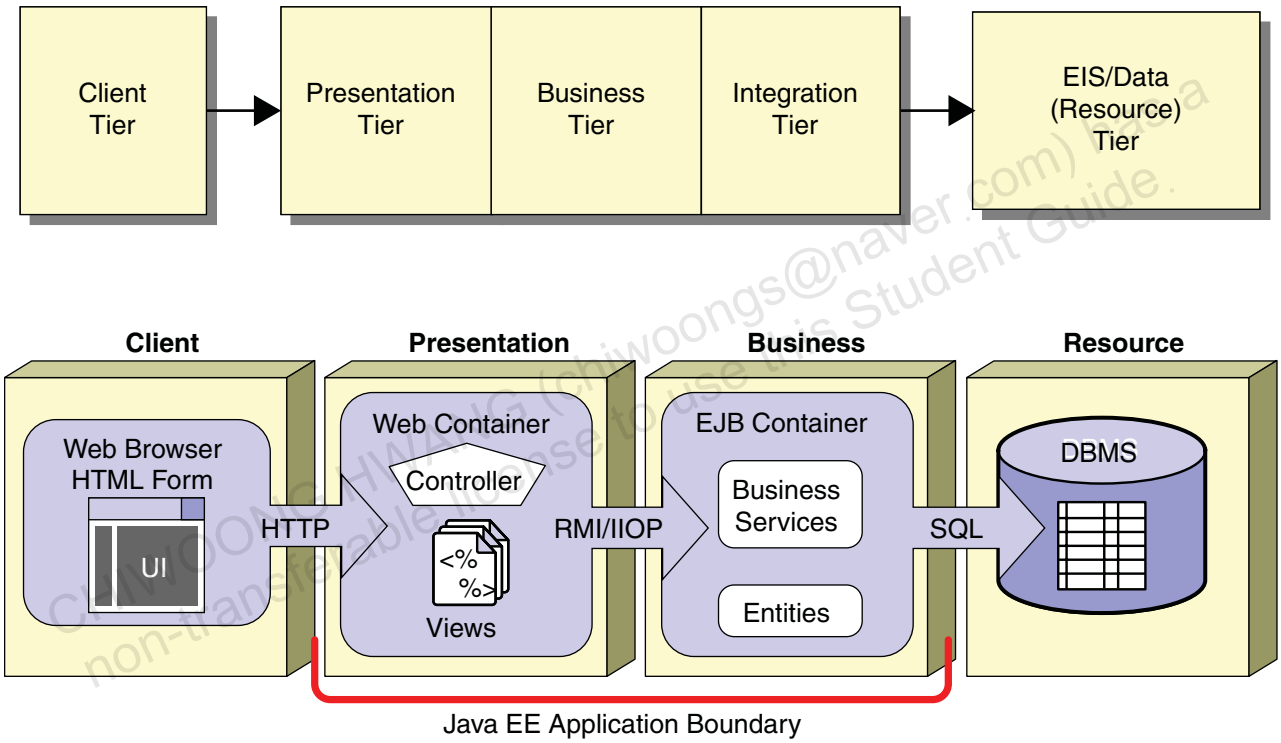


Figure 1-10 Java EE Tiered Architecture

In this example, the Java EE web container hosts the presentation-tier elements and the EJB container hosts the business-tier elements and integration-tier elements that interface with the database management system (DBMS). The Java EE technology elements can be configured to support many application architectures. Each architecture provides a framework that best supports specific application categories.

Java EE Application Architectures

There are several basic Java EE application architectures. The four most common types are:

- Web-centric architecture
- Combined web and EJB component-based architecture, sometimes called EJB component-centric architecture
- Business-to-business (B2B) application architecture
- Web service application architecture

Web-Centric Architecture

Figure 1-11 illustrates the web-centric configuration of the Java EE technology application server (Java EE server) middle tier.

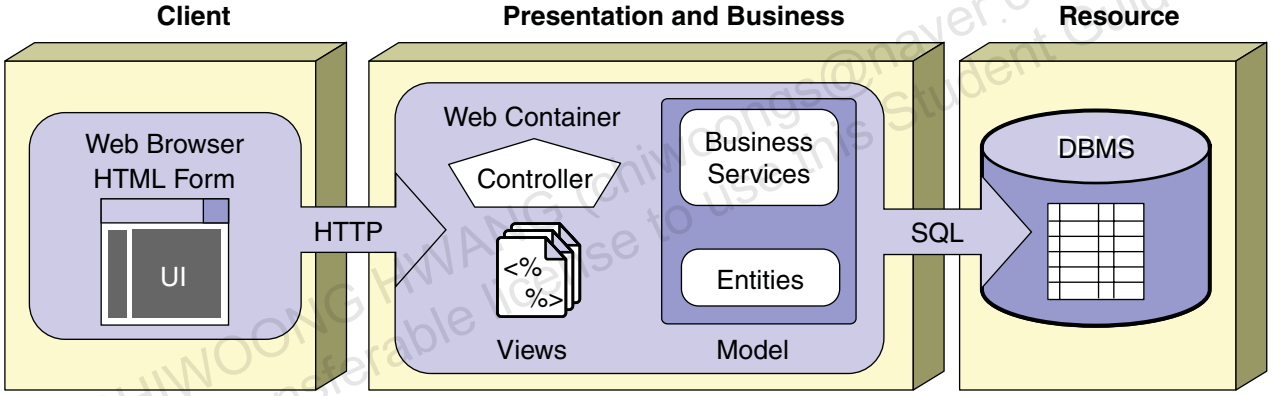


Figure 1-11 Java EE Web-Centric Architecture

This configuration uses only the web container of the Java EE technology server. The web container hosts all of the components that are required to generate the client view, process the business logic, and connect with the back-end data store.

Note – The introduction of EJB Lite in Java EE 6 allows the use of some EJB technology in web-centric architectures.

Combined Web and EJB Component-Based Architecture

Figure 1-12 illustrates the EJB component-based configuration of the Java EE technology application server middle tier.

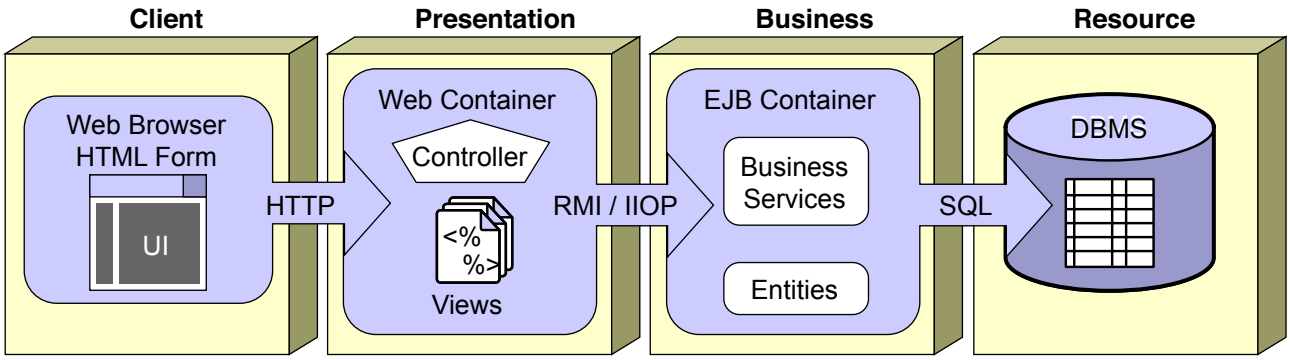


Figure 1-12 Java EE EJB Component-Centric Architecture

This configuration uses both the web container and EJB containers of the Java EE server. In this model, the business logic and data access components are located in the EJB container. The web-tier components process the incoming request and generate the view based on the results of the business process.

Java EE Profiles

Java EE 6 adds the standardization of profiles. As defined in the Java EE 6 specification, “A profile is a configuration of the Java EE platform targeted at a specific class of applications”. Many Java EE applications will never use all the features an enterprise application server. Application server vendors have provided proprietary configuration options to selectively enable and disable application server features. The problem with this proprietary approach is that application developers can not rely on specific features being enabled and therefore have difficulty maintaining application portability.

One profile has been created for inclusion in the Java EE 6 platform, the Java EE 6 Web Profile. The Java EE 6 Web Profile specifies a subset of the full Java EE platform. The web profile requires the following APIs:

- Servlet 3.0
- JavaServer Pages (JSP) 2.2
- Expression Language (EL) 2.2
- Debugging Support for Other Languages (JSR-45) 1.0
- Standard Tag Library for JavaServer Pages (JSTL) 1.2
- JavaServer Faces (JSF) 2.0
- Common Annotations for Java Platform (JSR-250) 1.1
- Enterprise JavaBeans (EJB) 3.1 Lite
- Java Transaction API (JTA) 1.1

Enterprise JavaBeans (EJB) 3.1 Lite

EJB 3.1 Lite defines a subset of the EJB 3.1 API that can be used by applications that do not require the full Java EE 6 Platform. EJB 3.1 Lite is supported through an embedded EJB container. The embedded EJB container allows the use of the commonly used EJB APIs inside of a web container or any Java SE application.

EJB 3.1 Lite is covered in more detail in ‘‘EJB Lite’’ on page 7-18.

B2B Application Architecture

Figure 1-13 illustrates the Java EE technology B2B application architecture.

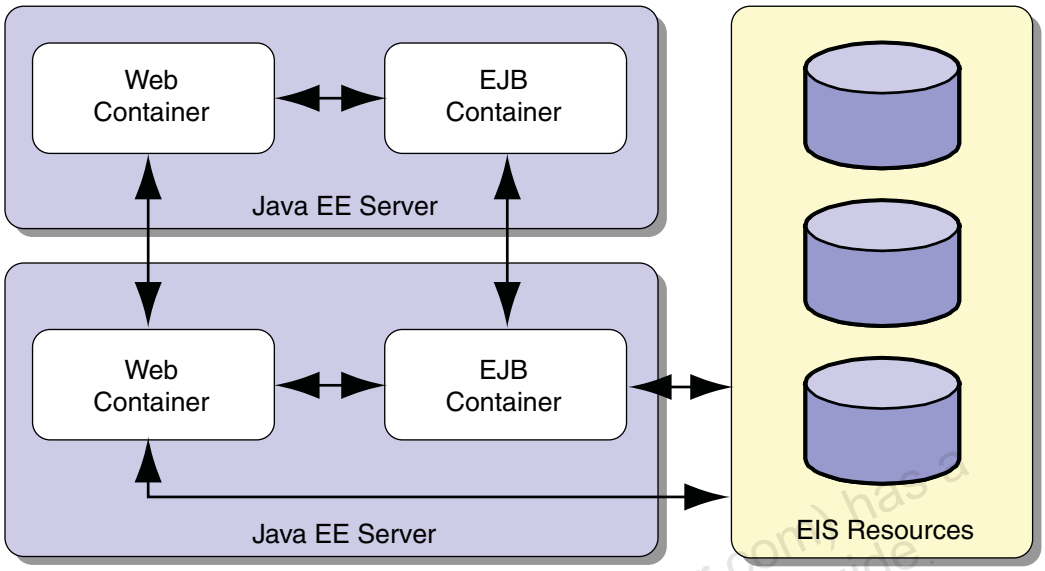


Figure 1-13 B2B Application Architecture

The B2B application architecture is an extension of the Java EE technology EJB component-based architecture. It involves two EJB servers, one in each business location. Each Java EE server hosts a web container and an EJB container.

This architecture allows for peer-to-peer communications between the corresponding containers in the two Java EE servers. The two web containers communicate using XML messaging over HTTP. This communication is loosely coupled. Java EE technology components in the two EJB containers communicate directly with one another. This communication is tightly coupled.

Web Service Application Architecture

Starting with the Java™ 2 Platform, Enterprise Edition (J2EE™) 1.4 platform, a Java EE component developer can expose the functionality of Java EE business components using web service technology. Figure 1-14 illustrates the Java EE web service application architecture.

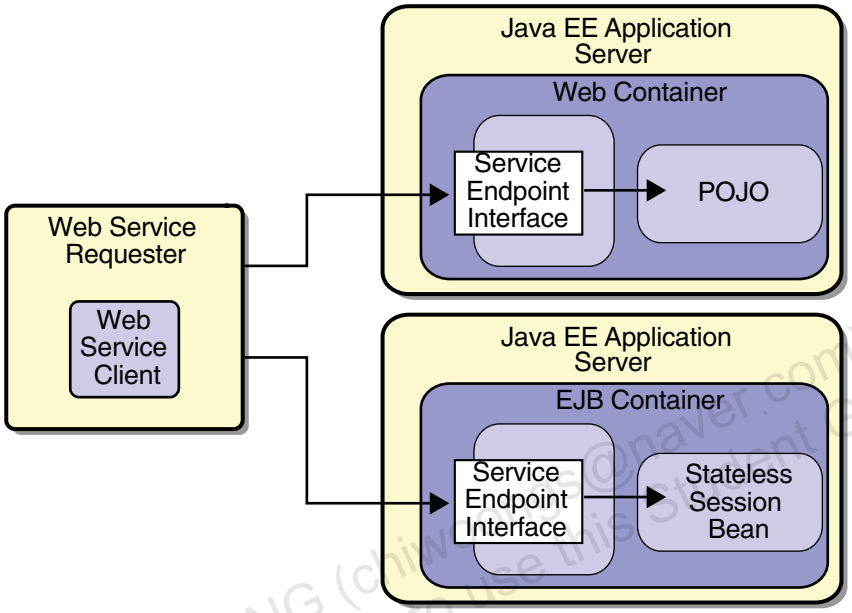


Figure 1-14 Java EE Web Service Architecture

Note – A web service registry is an optional component of a web service architecture.

The web service model for the J2EE 1.4 platform uses a stateless session EJB component as the web service endpoint, which serves as the access point to the service functions. The Java EE 5 platform defines a web service endpoint class that can be extended to create a web service endpoint. Communication between a web service client and the server-side endpoint is typically performed using SOAP-formatted messages that are transmitted over HTTP. The functionality that is exposed by a web service is typically defined by a web services description language (WSDL) file. The Java EE 6 platform adds support for RESTful web services with the introduction of JAX-RS. Web services can support both B2B and business-to-consumer (B2C) interactions.

Java EE Patterns

In addition to the basic architecture types that are supported by the Java EE platform, a set of architecture patterns exist to help the Java EE developer deal with some of the issues and complexities that are involved in creating a Java EE application.

Java EE Pattern Catalog

Architectural patterns provide standard solutions for well-understood programming problems. The Java EE pattern catalog contains architectural patterns that focus on the creation of scalable, robust, high-performance, Java EE technology applications. These patterns presuppose the use of the Java programming language and the Java EE technology platform, and in many cases, they are closely related to the Gang of Four (GoF) patterns.

Note – The Gang of Four is a common term used as shorthand for the four authors who wrote one of the most popular books on the topic of software patterns. The book, named *Design Patterns*, is listed in ‘‘Additional Resources’’ on page 1-2.

As illustrated in Figure 1-15, within the Java EE pattern catalog, the patterns are categorized according to the tier in which they are applied.

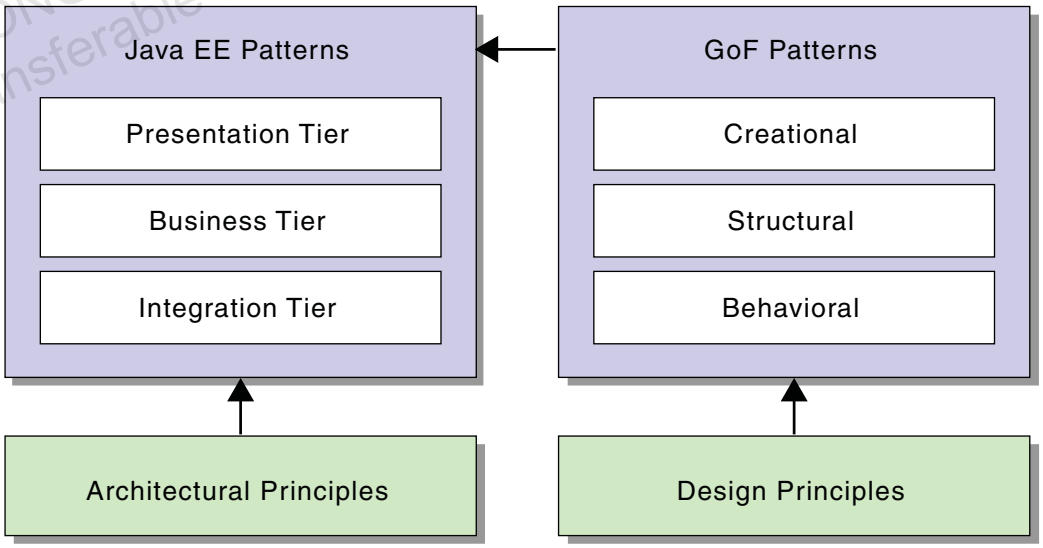


Figure 1-15 Java EE Pattern Tiers

The three tiers for which patterns are described are the presentation tier, the business tier, and the integration tier. In the same way that the GoF patterns were derived from a set of basic design principles, the Java EE patterns are derived from a set of standard architectural principles.

The description of each pattern includes at least a statement of the problem that the pattern is intended to solve. You can use a single pattern to solve a single problem. However, in many cases, a developer will face more than one problem within a single system. It is reasonable to expect that a developer will use more than one pattern within the overall system.

Application of Java EE Patterns

Figure 1-16 shows how a Java EE application developer might apply a set of patterns to a Java EE application.

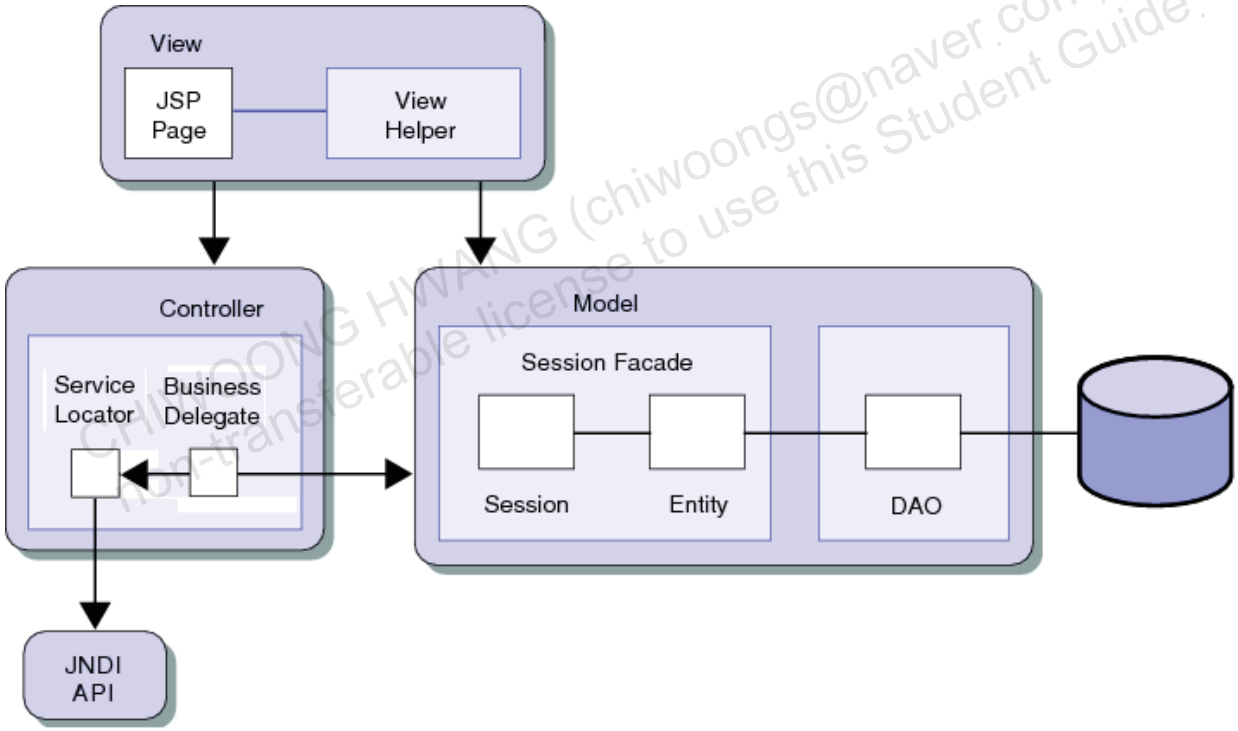


Figure 1-16 Using Java EE Patterns

The example in Figure 1-16 on page 1-24 uses the following Java EE patterns:

- Service Locator – A business tier pattern that abstracts components from the mechanism that is required to look up and connect to remote objects
- View Helper – A presentation tier pattern for a component that provides ancillary functions, such as constructing an intermediate data model, that are required by view components
- Session Facade – A business tier pattern that exposes the business functions that are implemented as coarse-grained services by the business logic components of the EJB tier
- Data Access Object (DAO) – An integration tier pattern for creating a component that encapsulates the data access code that is required to interact with a data store

You must be a little cautious in your use of patterns, however, because using too many patterns has the potential to unnecessarily complicate a system. However, it is often better to have a complicated system that uses clearly specified and well understood patterns, than to have a complicated system that is built without patterns. You will implement several Java EE patterns in the lab exercises for this course.

To find more information about Java EE patterns, use the resource suggestions that are listed in ‘‘Additional Resources’’ on page 1-2. For a more thorough description of software design patterns that you can use to effectively solve complex business problems with the Java EE platform technology, refer to the SL-500, *Java EE™ Patterns* course.

Java EE BluePrints

The Java software group has developed a set of guidelines and sample applications that a Java EE application developer can reference when designing and developing Java EE applications or application components. Known as the *Java BluePrints Solutions Catalog for Java EE*, this material is available online. See the ‘‘Additional Resources’’ on page 1-2 for a link to the online material.

Summary

The Java EE platform is an architecture for implementing enterprise-class applications using Java and Internet technology. A primary goal of Java EE technology is to simplify the development of enterprise-class applications through a vendor-neutral, component-based application model.

At the heart of the Java EE platform is the Java Standard Edition (Java SE). The Java EE specification incorporates a suite of other technologies and specifications, in addition to those defined by Java SE, to provide a rich feature set and enhanced server-side functionality for enterprise application developers.

A set of specifications maintained as part of the Java Community Process (JCP) defines the roles and responsibilities of Java EE platform vendors, tools providers, and component developers. The Java EE APIs define the contract between the application component developer and the platform provider using the interface mechanism defined by the Java programming language.

Java EE application components reside in containers from which they obtain runtime support. Each container type provides a support infrastructure that is tailored to the specific needs of the respective component types. A key feature of the Java EE platform is the strict separation of the application components from the general services and infrastructure. One of the main benefits of the Java EE platform for the application developer is that it makes it possible for developers to focus on the application business logic, while leveraging the supporting services and platform infrastructure that is provided by the container and application server vendor. The Java EE containers have access to a range of important API-based services, as defined by the Java EE specification.

There are several basic Java EE application architectures. The four most common are: Web-centric, EJB-centric, B2B, and web services. In addition to the basic architecture types that are supported by the Java EE platform, a set of architecture patterns exist to help the Java EE developer deal with some of the issues and complexities that are involved in creating a Java EE application. The Java EE pattern catalog contains architectural patterns that focus on the creation of scalable, robust, high-performance, Java EE technology applications.

Module 2

Java EE Component Model and Development Steps

Objectives

Upon completion of this module, you should be able to:

- Describe the principles of a component-based development model
- Describe the asynchronous communication model
- Describe the process used and roles involved when developing and executing a Java EE application
- Compare the different methods and tools available for developing a Java EE application and related components
- Describe how to configure and package Java EE applications

Additional Resources

Additional resources – The following references provide additional information on the topics described in this module:

- "JSR-000316 Java Platform, Enterprise Edition 6 Specification," [http://jcp.org/en/jsr/detail?id=316], accessed 1 August 2009.
- "Java BluePrints, Enterprise BluePrints," [http://java.sun.com/blueprints/enterprise/], accessed 1 August 2009.
- Eric Jendrock, Debbie Carson, Ian Evans, Devika Gollapudi, Kim Haase, Chinmayee Srivathsa. "The Java EE 6 Tutorial," [http://java.sun.com/javaee/6/docs/tutorial/doc/], accessed 1 August 2009.
- "NetBeans Documentation, Training & Support," [http://www.netbeans.org/kb/], accessed 1 August 2009.
- Fowler, Martin. "Inversion of Control Containers and the Dependency Injection pattern," [http://martinfowler.com/articles/injection.html], accessed 1 August 2009.

Principles of Component-Based Development

Component-based development is a key feature of the Java EE platform. Java EE components are meant to be both portable and reusable. Java EE application components might come from a variety of sources both internal and external to the development team. The EJB specification was designed from the outset to support the assembly of applications using components from different vendors. Components can be authored without detailed knowledge of the environment in which they will be used or of the other components with which they will interact. This principle is called *loose coupling*. Loosely coupled systems are easier to test and maintain, and components of a loosely coupled system are easier to reuse.

Java EE Components

A component in the Java EE platform is not necessarily a class, but is more commonly a grouping of classes and interfaces that implement a self-contained unit of functionality.

The Java EE model defines a number of different component types, each of which is tailored to fulfill a specific need. The following figure illustrates the primary Java EE component types in their respective containers. The Java EE component types include session beans, entity classes, message-driven beans, servlets, and components that are based on JSP technology.

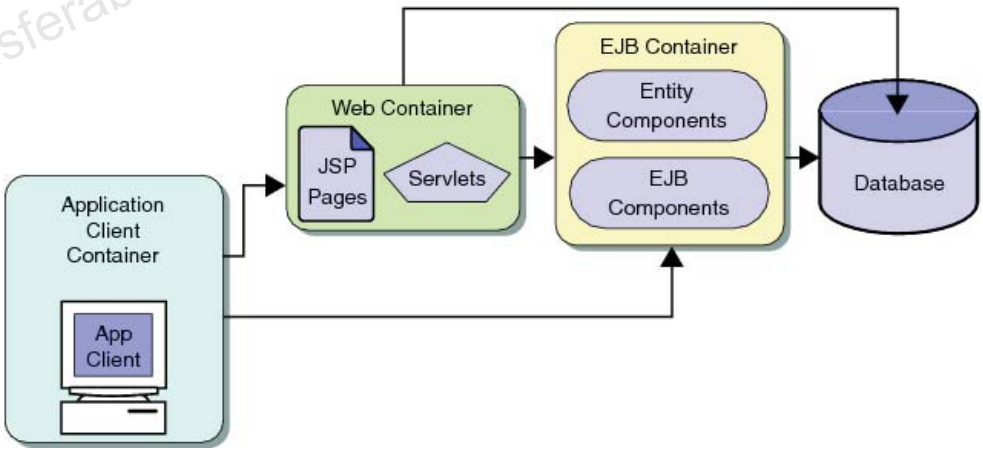


Figure 2-1 Java EE Components

Java EE Component Characteristics

Some of the component types have subtypes. Components also have some or all of the following essential characteristics, depending on the component type:

- State and properties
- Encapsulation by a container
- Support for local and distributable component interactions
- Location transparency
- Component references that are obtained using a naming system

The following information describes the component-based application framework that is supported by the Java EE specification.

Component State and Properties

The manner in which a component manages state or defines properties can have implications on its performance or on how it can be used within the Java EE framework.

Component State

Components might have *state*. State is associated data that must be maintained across a number of method calls from the same client. However, there are certain performance advantages that follow from the assumption that some components have no state. Specifically, the component infrastructure can cycle a client's method calls to different instances of the component on different hosts. This cycling is advantageous for load balancing and fault tolerance. Components might be designed to be intrinsically stateless or stateful. Other component types might have both stateless and stateful forms.

Component Properties

A *property* is some feature of the component that can be either read *and* written by its clients or read *or* written by its clients. A property might be represented internally by an instance variable, but this is irrelevant to a component's clients. In the Java programming language, properties are typically modeled as *accessor* and *mutator* method pairs. For example, the `getName` and `setName` methods represent the `name` property.

Encapsulated Components

Encapsulation is an important concept in object-oriented programming, and the Java EE model takes encapsulation a step further than most programming models. Components are encapsulated by containers that both manage their life cycles, as well as isolate them from other components and from the runtime environment.

Component Proxies

A important feature of the Java EE component model is that some components are controlled through proxies. One component can only interact with another component through that component's proxy. Even if the components are located in the same Java Virtual Machine (JVM™), it is an error for one component to directly instantiate another component with the `new()` operator or to make direct method calls on another component's implementation. Interfaces are frequently used to define the functionality that will be made available by a component's proxy.

Figure 2-2 illustrates how Java EE application components interact through interfaces.

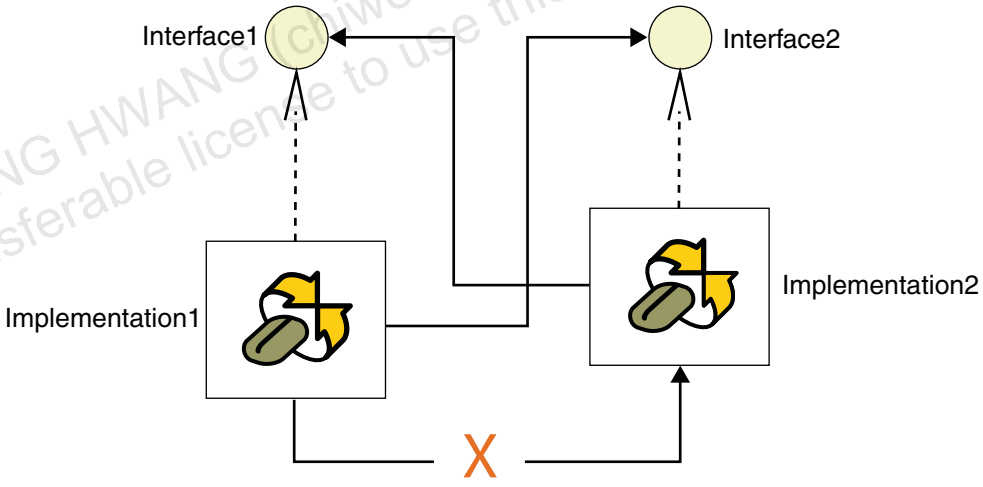


Figure 2-2 Interaction of Java EE Application Components Through Interfaces

Provided that the proxy implements the same interface as the real component, the client is unaffected by this imposition. Java EE 6 allows this proxy based access approach to be implemented for some components without requiring an interface. The use of proxies allows the component to be managed by its container, which has important advantages for you.

Distributable and Local Component Interactions

The Java EE component model supports both local and distributable component interactions for some component types. You specify whether an interaction between two components is to be local or distributable.

- If an interaction between two components is local, the application server typically makes the components available to each other in the same JVM.
- If an interaction between two components is distributable, the application server must provide an RMI infrastructure by which the two components communicate.

There are costs and benefits associated with both of these strategies.

Distributed Components and RMI

To support distributable component interaction, the application server must provide an RMI infrastructure to support communication. Because the Java EE component model allows a strict separation of interface from implementation, it is straightforward to provide an RMI infrastructure in a way that is mostly transparent to you.

As shown in Figure 2-3, `Component1` interacts with `Component2` through its interface, `Interface2`. When the components are distributed, `Interface2` is implemented by a *stub*. The stub contains the communication logic that allows it to make remote method calls. Similarly, `Component2` is unaware that its methods are not being called directly by `Component1`. In fact, the methods for `Component2` are called on behalf of `Component1` by a *skeleton*. The stub and the skeleton jointly implement the communications infrastructure.

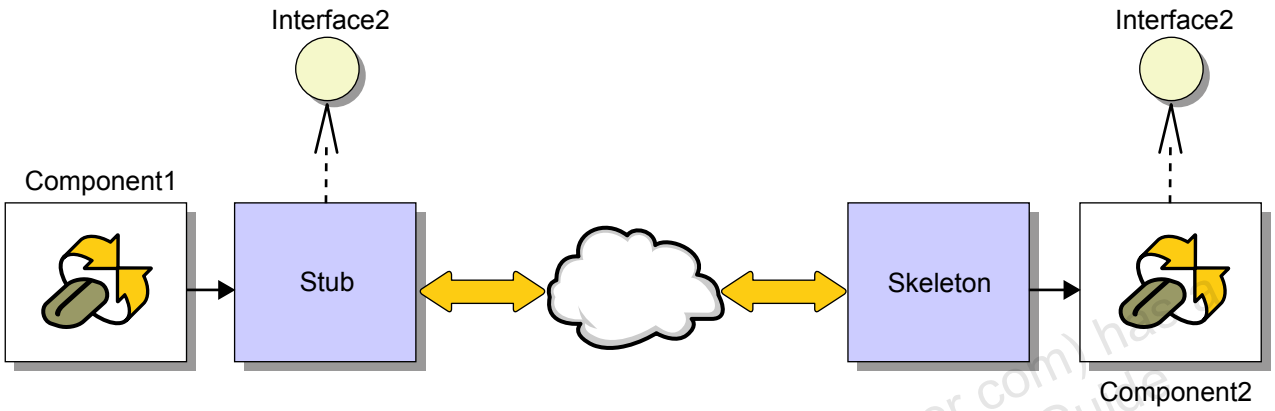


Figure 2-3 RMI Infrastructure for Distributed Components

The RMI infrastructure must manage the following design issues:

- *Marshalling and unmarshalling* of arguments and return values

  Java programming language serialization forms part of this process. However, serialization alone is not sufficient if the RMI infrastructure must be language-independent. Nevertheless, objects that are to be passed between distributed components must normally be serializable

- *Passing distributed exceptions*

  When the components are distributed, error conditions might arise that are independent of the application logic itself, such as a network disconnection. Normally, the caller can expect to catch a `java.rmi.RemoteException` in such circumstances.

- *Passing security context and transaction context* between the caller and the target

  Security and transaction management are presented later in the course. For now, you should understand that there is information to supply, along with the method call that is distinct from arguments and return types. For example, the target component might need to ensure that the caller has the access rights to invoke the method.

The RMI protocol that an application server must support is IIOP, which is part of the CORBA specification set. IIOP makes applications that are based on EJB components that are interoperable with CORBA services. However, there is nothing to prevent a server vendor from supporting protocols in addition to IIOP. For example, some vendors support RMI with Java Remote Method Protocol (JRMP) as well.

Calling Semantics for Local and Distributable Components

Local and distributable component interactions follow different calling conventions:

- Local components interact using the ordinary Java programming language calling convention. Arguments are passed in shared memory, which means that an instance of an object that is passed as an argument from one component to another component can be modified by the target component. The modifications are visible in the calling component. In other words, the caller and the target both see the same instance of the argument.
- Arguments that are passed as part of a distributable component interaction, on the other hand, are passed by copying their state. Arguments are serialized and then transferred over the network from the caller to the target, which means that the target cannot modify the caller's instance of the argument.

There is one exception to these rules. Arguments that are themselves distributable components are always passed by passing a stub, regardless of whether the components are local to one another. Because a stub is passed, the object to which it is passed can call methods over the wire onto the caller's instance of the object.

Costs and Benefits of a Distributed Component Model

The costs of distributed components derive primarily from RMI overheads. The marshalling and unmarshalling of the arguments and return values can be a slow process. Each method call requires a network call setup, and then the data is carried over a network, which is a shared resource. In summary, the decision about whether to use distributable or local interactions must be carefully weighed in each case.

The benefits of distributed components derive mainly from the location transparency that the distributed model provides, including increased fault tolerance and improved load sharing.

Note – Fault tolerance and load sharing between servers may be achieved with local components by using the advanced features of an application server such as Glassfish. Load balancing, session replication, and application server clustering do not require remote components. Check the advanced features of your application server as these features are not required by the Java EE specification.

Location Transparency

Location transparency is one of the design goals of the distributed component model of the Java EE platform. Location transparency means that in any component-to-component interaction, the calling component is not concerned with the physical location of the target component. In practice, in a large system, each component might be deployed in more than one host, which has the following important benefits:

- Load balancing – Calls can be cycled between hosts to share the overall load.
- Fault tolerance – In the event of a failure, calls can be directed to hosts that are still in service while failed hosts are repaired.

It is the responsibility of the application server vendor to realize these benefits. Your responsibility is to create components that comply with the specification, which automatically results in the benefits of location transparency.

Naming Services in the Component Model

In the Java EE component model, some components can interact only with each others' interfaces. Therefore, a component requires a way to get a reference to another component's interface without making use of the actual implementation. The facility to get a reference to another component's interface is provided by a naming service. An application server provides its components with a central repository of components that are accessible by name.

Figure 2-4 shows the process of using the naming service to look up reference information.

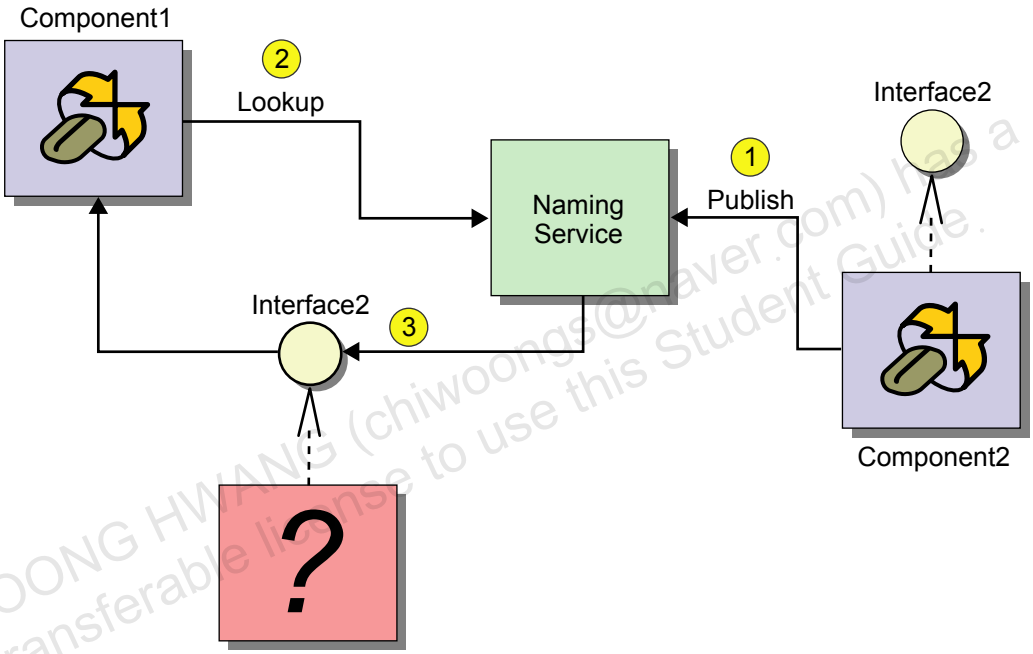


Figure 2-4 Naming Services in the Component Model

The three steps in Figure 2-4 are described as follows:

1. When the application is deployed, all of the components that need to be made accessible to clients are registered with the naming service. In the example, the component called `Component2` is published. This implements a Java programming language interface, called `Interface2`.

2. The component called `Component1` can only make method calls on `Component2` through `Interface2`. To get a reference to `Interface2`, `Component1` performs a lookup using the central naming service that is based on the logical name of `Component2`.
3. The naming service supplies an instance of a component that implements `Interface2`. The object that is supplied is not necessarily an instance of `Component2`. The object might be a proxy for `Component2`, but this is invisible to `Component1`.

   The `Component1` can then call methods on `Component2` using `Interface2`.

Use of the Java Naming and Directory Interface™ (JNDI) API in the Java EE Component Model

Java EE applications use the JNDI API as a general name lookup service to locate components, external resources, and component environment properties. The JNDI API abstracts the detail of the underlying naming protocols and implementation.

The use of the JNDI API as a general name lookup is similar to its use for communicating with an external naming service, such as the directory server. However, only a subset of the full API is required when JNDI API is used in the context of a Java EE application.

In practice, many application servers use a CORBA-compliant RMI infrastructure, and the implementation that underlies the JNDI API is a CORBA naming service. Alternatively, the implementation might use LDAP, or something entirely proprietary. In most cases the protocol has no effect on the application developer, because the API conceals these details.

The `Context` Interface and the `InitialContext` Object

The `Context` interface (`javax.naming.Context`) is the basis for all naming operations. The `InitialContext` object is a specific implementation of the `Context` interface, and represents the entry point to the naming service. A JNDI API operation typically begins by instantiating the `InitialContext` object.

The namespace that is accessible by way of the JNDI namespace can be hierarchical. A `lookup` operation on an instance of a class that implements the `Context` interface results either in an object or in a subcontext. The subcontext also implements the `Context` interface and can, therefore, form the basis of a new lookup. Code 2-1 shows two code snippets that have the same effect in practice.

Code 2-1 Performing a Lookup Operation

```
1   Context c = new InitialContext();
2   Object o = c.lookup("aaa/bbb");
```

Or

```
1   Context c = new InitialContext();
2   Context subcontext = (Context) c.lookup("aaa");
3   Object o = subcontext.lookup("bbb");
```

All JNDI API operations throw a `NamingException` in the event of a problem, so this exception must be handled in the code.

Configuring the `InitialContext` Object

When JNDI API is used within a component, the container must provide configuration to the `InitialContext` class. You just use the default constructor of the `InitialContext` class to create `InitialContext` as follows:

```
Context c = new InitialContext();
```

When you use the JNDI API outside of the context of an application server, you may be required to supply configuration information. This information varies from one naming service to another, but there are usually two pieces of required information:

- The name of a class that implements the underlying naming protocol
- A Universal Resource Locator (URL) that specifies the location of the naming service

This information is typically supplied in the form of name-value pairs that are passed in a `Hashtable` object to the `InitialContext` constructor or as a properties file that is read automatically in the `InitialContext` constructor.

Code 2-2 shows a example of how to configure a naming context for an application server.

Code 2-2 Configuring a Naming Context

```
1   Hashtable env = new Hashtable();
2   env.put ("java.naming.factory.Initial",
3      "com.sun.jndi.cosnaming.CNCtxFactory");
4   env.put("java.naming.provider.url","iiop://hostname:3700");
5   Context c = new InitialContext(env);
```

Keep in mind that this configuration is only required for access to JNDI when using a client component located outside the Java EE application server host. A properties file may be used in-place of programmatic configuration.

Using JNDI API as a Resource Locator

Within the environment of a component, you can use JNDI API calls to locate objects other than other components. For example, you can use JNDI API calls to locate the following objects:

- Connections to relational databases
- Connections to messaging services
- Message destinations
- Component environment variables
- Connections to legacy systems that are supported by resource adapters

Narrowing and Remote Objects

For all lookups that use the JNDI API, the result of the `lookup` method is cast to the appropriate type. For example, connections to relational databases are obtained in the form of `DataSource` objects (`javax.sql.DataSource`). Code 2-3 shows the code that is required to look up a database resource named `jdbc/bank`.

Code 2-3 Looking Up a Data Resource

```
1   Context c = new InitialContext();
2   DataSource ds = (DataSource)c.lookup("jdbc/bank");
```

If the object that is returned by the lookup is not of the expected type, the cast fails and the caller gets a `ClassCastException`.

Additionally, for remote objects, the result of the `lookup` requires *narrowing* to the appropriate type. This narrowing is to ensure future compatibility with a broad range of RMI infrastructures. Some application servers allow remote objects that are looked up with JNDI to be cast. Code 2-4 shows narrowing.

Code 2-4 Narrowing a Lookup Return Value

```
1   Context c = new InitialContext();
2   Object o = c.lookup("ejb/BankMgr");
3   BankMgr bankMgr = (BankMgr)
4        javax.rmi.PortableRemoteObject.narrow
5            (o, BankMgr.class);
```

Remember that when the client of a Java EE component asks for a local reference instead of a remote reference, the result of the JNDI API lookup is not a remote object and will never require narrowing.

Using a Component Context to Locate Components

Java EE EJB components typically have a reference to an `EJBContext` object that provides access to environmental data and manipulation, such as security information, transaction control, and resource lookup. Because an `EJBContext` object is used to provide several critical pieces of information, one is usually already present in an EJB component.

The `EJBContext` object works in a similar way to a JNDI Context and can be used in place of a JNDI Context if desired. The `EJBContext` object actually performs JNDI lookups within the `java:comp/env` JNDI Context and can be used to locate EJB references, DataSources, and any other resource reference that can be stored in JNDI. Using an `EJBContext` object removes the need to repeatedly call a new `InitialContext` method and simplifies exception handling by only causing runtime exceptions.

An `EJBContext` object can be used to look up EJB 2.1 Home references. Even when looking up EJB 2.1 Remote Home references using the component context, the use of `PortableRemoteObject.narrow` is not required.

Code 2-5 Using an `EJBContext` for Lookups

```
1   import javax.ejb.*;
2   import javax.annotation.*;
3
4   @Stateful
5   public class MySessionBean implements MyInterface {
6
7     @Resource private javax.ejb.SessionContext context;
8
9     public void myMethod() {
10       BankMgr bankMgr = (BankMgr)context.lookup("ejb/BankMgr");
11    }
12
13  }
```

Using Dependency Injection to Locate Components

Dependency injection is not a new concept, previously the design pattern name for dependency injection was Inversion of Control. The idea behind dependency injection is to remove the need for the developer to repeatedly write resource lookup code. Resource lookup in Enterprise Java is typically done with JNDI. Dependency injection eliminates JNDI coding. Dependency injection was limited to being used within certain types of components in Java EE 5. Java EE 5 applications sometimes required a mixture of dependency injection and JNDI API usage. Java EE 6 enhances dependency injection support, thus eliminating the need for almost all JNDI use by application developers.

Dependency injection requires the use of Java Annotations. Java Annotations are a new feature to the Java SE 5 platform. Annotations provide a way to embed extra data about classes, variables, and methods in source code and also in compiled class files. This extra data added to class files can be ignored by the Java Runtime Environment if it is not needed. Software, such as a Java EE 6 application server can check for the presence of annotations using the Java reflections package.

To implement dependency injection, an application server assigns a reference value to a variable before it can be used. To guarantee the application server has a chance to perform the injection, a component must have its lifecycle controlled by the server. These types of components are referred to as managed components and include Servlets, JSPs, and EJBs. Plain Java classes still require developers to write JNDI lookup code.

In a previous version of Enterprise Java, developers and deployers were required to create and edit large amounts of XML deployment descriptors. In the Java EE 5 platform, annotations and dependency injection replaced the need for most deployment descriptor information.

Code 2-6 Dependency Injection as an Alternative to Lookups

```
1   import javax.ejb.*;
2
3   @Stateful
4   public class MySessionBean implements MyInterface {
5
6      @EJB private BankMgr bankMgr;
7
8   }
```

The Asynchronous Communication Model

There are two basic ways that a component can interact with other components, synchronously or asynchronously. Although each style of interaction has its place in the Java EE programming model, the asynchronous model, in some cases, provides distinct advantages.

Comparison of Synchronous and Asynchronous Component Interactions

Synchronous component interactions, whether distributed or local, follow basic request-response semantics. Communication between a web browser and web component over HTTP uses request-response semantics. Synchronous interactions are characterized by the fact that the calling component blocks until the operation completes and the calling component receives a direct response. Synchronous interactions might not be suitable for operations that take an extended time to complete or that require a high guarantee of service.

Asynchronous interactions use request-notification semantics. That is, the caller makes a request and then continues processing. The caller does not block while it waits for a notification. Instead, the caller might receive a notification at some later time. Request-notification semantics allows an application to support operations that can take a long time to complete, and also reduces *coupling* between components.

Asynchronous Component Interaction

Figure 2-5 shows an example of an application that uses asynchronous component-to-component interactions to support a legacy system.

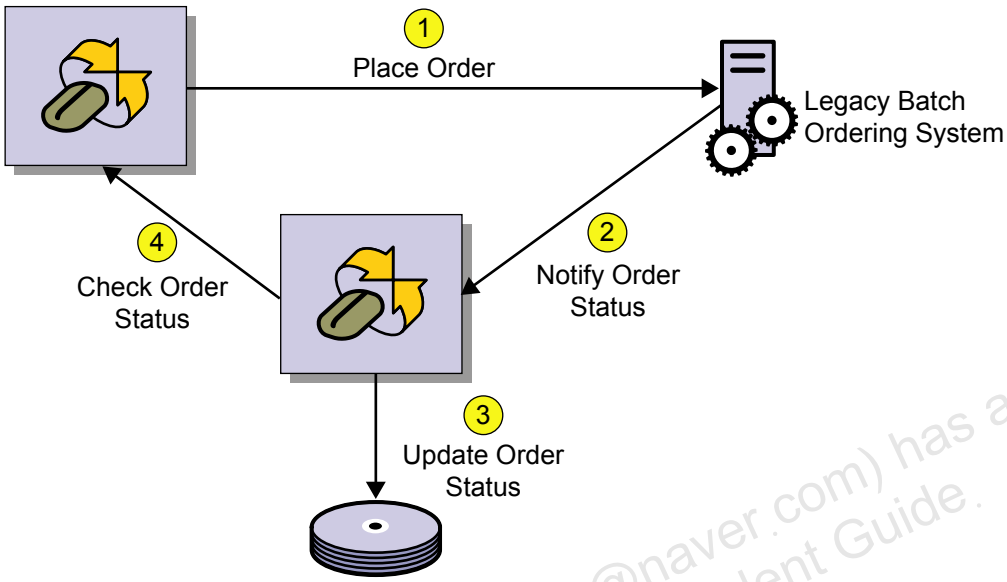


Figure 2-5 Asynchronous Component-to-Component Interaction

The steps in Figure 2-5 can be described as follows:

1. A component places an order on a legacy batch processing system on behalf of a client. The legacy system processes orders in batches of one hundred, so it might take several hours for the order to be processed.
2. When the batch processing system has completed the order, it notifies another component.
3. This component updates a database that contains order status information.
4. If the client requests information about the status of the order, a component can check the order status database. This last step can be performed using a synchronous interaction, asynchronous messaging, or an email notification.

Asynchronous Messaging

The J2EE platform 1.3 required an application server to provide a messaging service that supported asynchronous component interaction. In addition, beginning with the J2EE platform 1.4, servers must provide the infrastructure for XML messaging based on the web services model.

Components use the JMS API to send messages to other components or to external resources. Message-driven beans act as consumers of messages either from other components or from external resources.

Java EE 6 supports asynchronous processing without requiring the developer to code an asynchronous messaging solution.

The role of messaging and message-driven beans is presented later in this course.

Benefits and Costs of Asynchronous Interactions

The use of asynchronous component interaction offers two main benefits:

- Reduced coupling between components

  This benefit results in the reduced long-term costs of software management. Reduced coupling between components is a standard principle of software engineering, because the fewer dependencies that exist between components, the easier it is to manage components independently. An asynchronous model inherently offers loose coupling.
- Accommodation of operations that take an extended time to complete

There are two main costs associated with asynchronous component interactions when compared to synchronous component interactions:

- More complex infrastructure requirements
- Less efficient use of network resources

Developing Java EE Applications

Java EE application development is typically performed by a group of people, each with separate roles and responsibilities, such as system architects, component developers, and application assemblers.

Java EE Roles

The Java EE specification defines several roles that are involved in the development, deployment, and management of an enterprise application. Although most organizations have strategies for dividing up the work between different members of a project team, the Java EE specification goes as far as mandating, to a degree, how this is to be done. Java EE roles include:

- Application component provider

  The *application component provider* develops Java EE components. These components can include EJB components, web components, and possibly resource adapters. The output of this role is compiled classes and *XML deployment descriptors*. The descriptors might be incomplete with respect to the finished system. The output from this process can be used on any Java EE platform-compliant environment. Application component providers produce vendor-neutral output.

- Application assembler

  The *application assembler* takes completed Java EE components and assembles them into a deployable application. The components themselves can be from various sources, including sources from outside of the organization. The application assembler resolves cross-references between components and configures the components to suit the application as a whole. Because of simplified packaging rules and support from tools this role has become less critical.

- Deployer

  The *deployer* is responsible for the deployment of the assembled application in a specific operational environment. The deployer is responsible for resolving references to external resources, configuring the run-time environment of the application, and integrating the application with the security infrastructure.

- System administrator

  The *system administrator* maintains and monitors the application server environment and ensures that optimum performance is achieved. For example, this role might involve making decisions that are related to load balancing and redundancy.

- Tool provider

  The *tool provider* implements development, packaging, assembly, and deployment tools.

- Product provider

  Product providers are vendors of application servers and the supporting hardware and software for those servers.

For the purposes of this course, two role distinctions are important:

- The distinction between the tool provider and the product provider

  Most vendors of applications servers that are based on Java EE technology supply assembly and deployment tools with their products. However, the standards-based nature of the Java EE specification allows a developer to use tools from third-party tool providers if they so desire. Application components that are developed and assembled according to the Java EE specification should run on any compliant platform.

- The distinction between the component provider, application assembler, and deployer

  This distinction is operative throughout the Java EE specification, and is fundamental to this course as well. Remember that although the same person can be a developer, assembler, and deployer, these are separate roles with defined responsibilities. For example, the component provider might use an environment variable within the application code that the deployer sets at deployment time.

Note – In the lab exercises for this course, you will sometimes serve in the role of the component provider, application assembler, and deployer, because developing Java EE software requires completion of several distinct steps that involve one of more of these roles.

Steps for Developing a Java EE Application

Developing a Java EE application typically requires performing the following tasks:

- Designing

  As in any project, design is one of the most critical steps. Simply using the Java EE application framework resolves many design issues, such as how to apply security or transaction management in an application. Java EE design typically involves a higher level of abstraction, such as deciding on what set of components to use to implement the presentation tier or the business logic.

  A system architect designs the application architecture and specifies component interfaces. A well-defined set of component interfaces at the design step can make all of the following steps go relatively smoothly. The steps required to design and architect a Java EE application are covered in SL425, *Developing Architectures for Enterprise Java Applications*.

- Coding

  In this step, component developers create the underlying implementations of the Java EE components. Developers code, compile, and debug the components to make sure that the components fulfill their interface contracts. Unit testing is an important part of any substantial development process. Coding can be done in any editor, and compiling and testing can be done either manually or with a tool.

  Java EE application components might be developed for a particular application, or they might be general-purpose components or part of a software library. Component developers typically code to an interface specification using logical names of other application components. Similarly, the component developer typically uses logical names for external resources, such as databases the component accesses at runtime. These logical names must be resolved when a component is deployed as part of a functioning application. This is typically done using a naming service.

- Creating deployment descriptors

  A component provider uses an XML deployment descriptor to list resources and other components that are accessed by a Java EE component, as well as the names used to reference them. Components that do not have pre-programmed connections to outside resources are reusable in many applications.

Ultimately, the application server creates resource connections and resolves references between components using the information contained within a deployment descriptor. You can create a deployment descriptor using a text editor or with a third party or vendor-supplied tool.

- Packaging

  A set of class files is not considered a component until it is packaged as a component. All pertinent files must be packaged together in an appropriate archive file. Packaging can be done either manually at the command-line or with a tool.

- Assembling

  In this step, the application components are assembled into a working application. Part of this process involves resolving references between components that are provided by different component vendors.

- Deployment

  The application must be deployed to the application server either manually or using a vendor's tool. This step typically involves some vendor-specific configuration of the application components or the runtime environment, such as load-balancing. Most server vendors provide a tool to accomplish deployment.

Figure 2-6 illustrates how application components are assembled into application modules that are then deployed as an application to an application server.

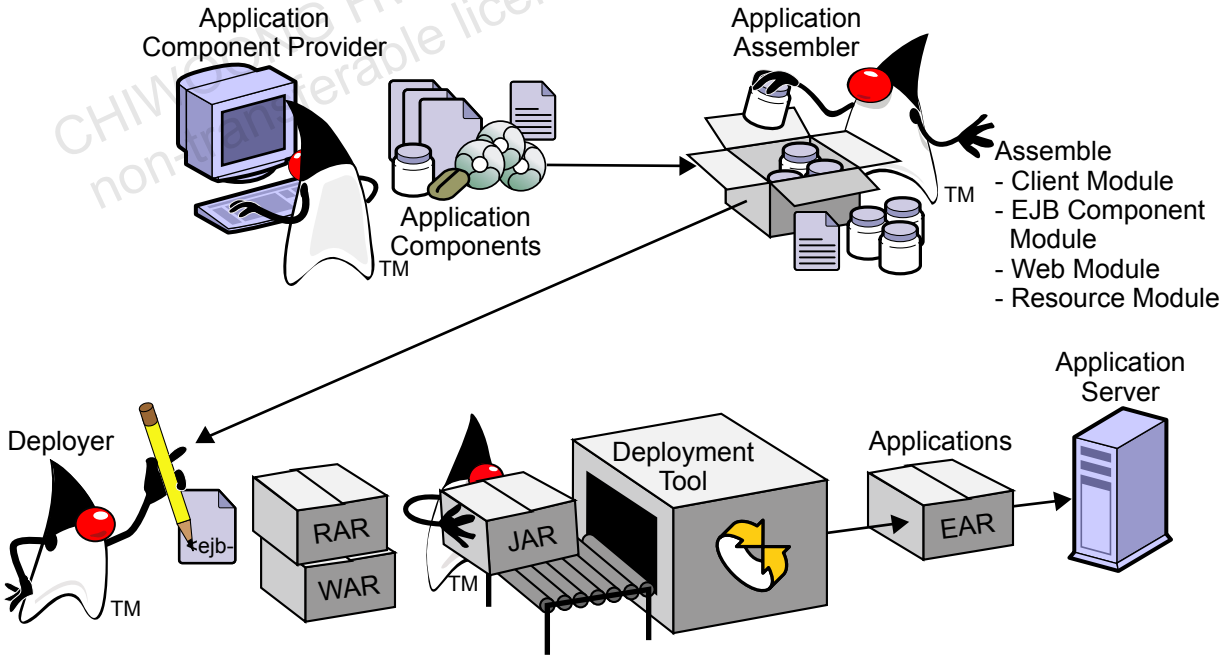


Figure 2-6 Java EE Application Development Process

Development Tools

You can create Java EE technology components using a variety of tools, such as text editors for coding and file editing, command-line compilers, deployment and assembly tools, or integrated development environments (IDEs) that manage the entire iterative development cycle under the umbrella of a single application.

IDEs provide an end-to-end environment for coding, packaging, and deployment. In some cases, IDEs provide an end-to-end environment for providing vendor-specific configuration of Java EE applications. IDEs typically have the following features:

- An editor
- The ability to manage Java EE components in a graphical manner
- The ability to compile from within the IDE
- The ability to debug source code
- The ability to edit deployment descriptors using a graphical tool
- The ability to deploy to one or more application servers

The Netbeans™ and Eclipse™ IDEs provides all of these features and more. IDEs may provide other features, such as database exploration, entity class creation from existing database structure, and advanced refactoring capabilities. Many IDEs are extensible using plug-ins, which provide potentially unlimited capabilities.

Configuring and Packaging Java EE Applications

In a large development project, a number of developers or development teams might contribute the components for a Java EE application. Developers prepare individual components for distribution by packaging components into archive files that contain the relevant class files and XML deployment descriptors. The format of the archive file and the information contained within the deployment descriptor depends on the component type. Component archives are assembled into application modules that are then packaged into a *super archive*, which is an archive of archives that forms the complete application. Because the contents and structure of the archive files are mandated by the Java EE specification, you can archive and bundle Java EE components using any Java EE platform-compliant archiving or assembly tool.

There are four basic types of archive files that are used in a Java EE platform development project:

- Web archive (WAR) files
- Java archive (JAR) files
- Resource archive (RAR) files
- Enterprise archive (EAR) files

Each type of archive file declaratively defines the characteristics and operating requirements of the archived components using a deployment descriptor.

Java EE application components are generally created in modules that serve as the basis for the construction of an archive file. A Java EE module consists of one or more Java EE components for the same container type and one component deployment descriptor of that type. For example, EJB components can be developed as part of an EJB module that is subsequently used to create an EJB JAR file.

Web Archive Files

Figure 2-7 shows the fundamental elements that constitute a web application.

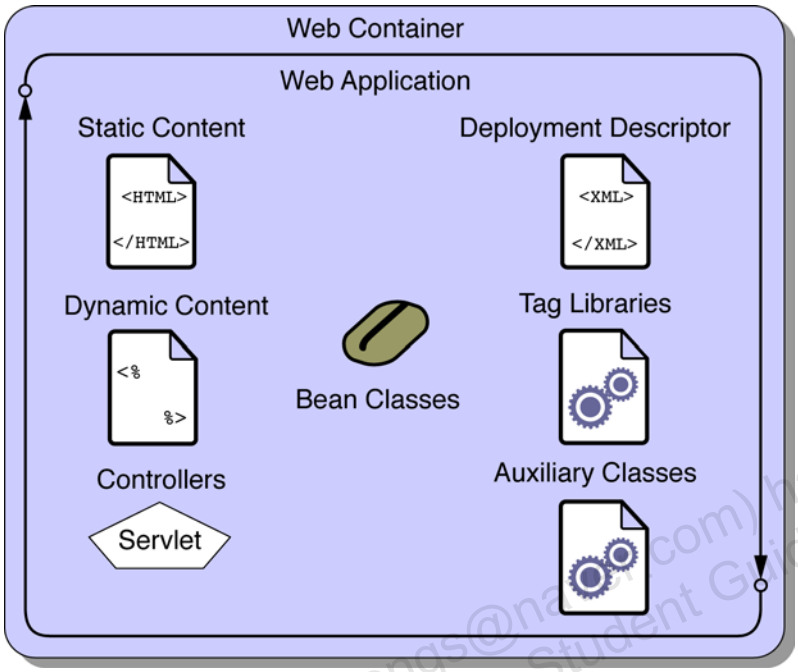


Figure 2-7 Web Application Elements

Although the diagram only shows one icon for each element or component type, there might be hundreds of individual elements that are contained within a web application.

Java EE 6 adds support for an EJB component to be packaged in a war file. This also eliminates the need for a super archive, known as an EAR.

As illustrated in Figure 2-8, the contents of a web application are stored as a web archive file for distribution and incorporation as part of a deployable Java EE application.

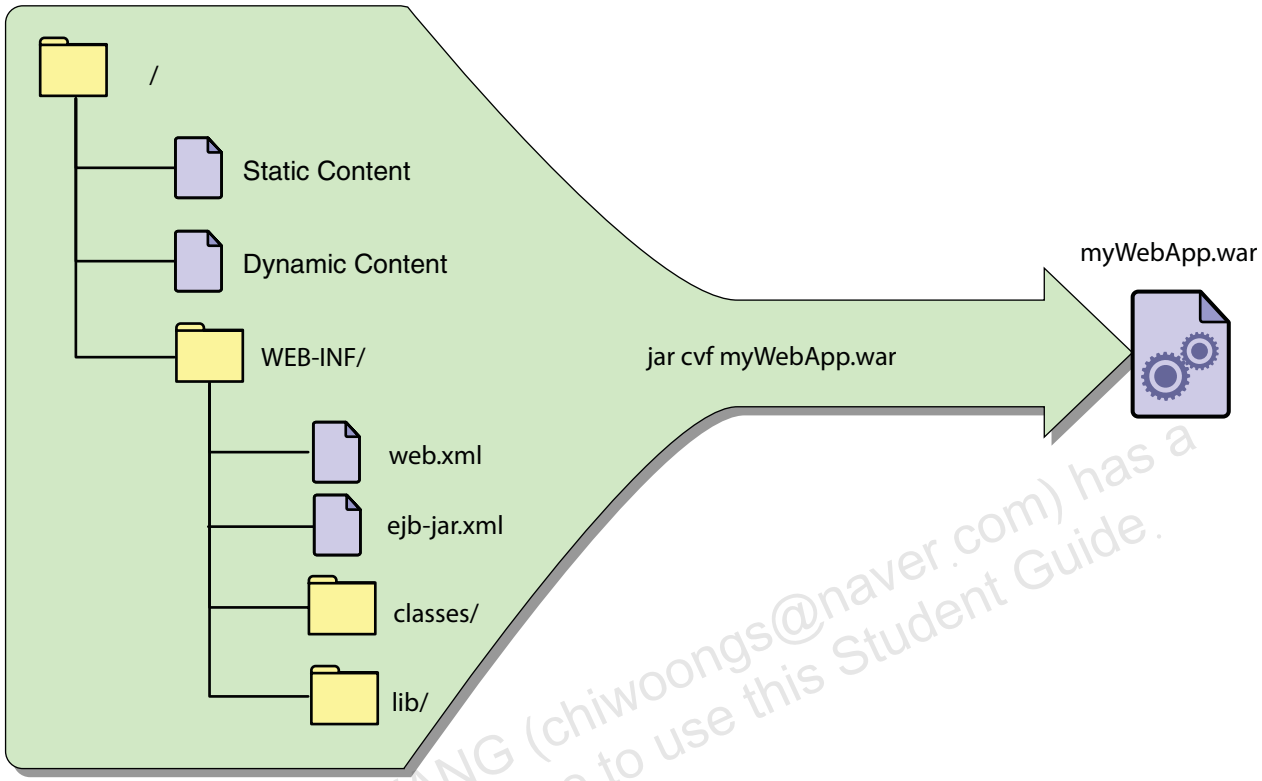


Figure 2-8 Web Archive File Creation

A web archive file uses a `.war` file extension.

In addition to allowing EJB components to be placed in the `classes` directory of a WAR, Java EE 6 allows library JAR files that contain EJB components to be placed in the `WEB-INF/lib` folder of a `.war` file. A library of EJB components is not considered a stand-alone enterprise module.

Java Archive Files

Java Archive (JAR) files provide a standard mechanism for packaging and distributing Java class files and related resources. JAR files are normally given names that end in `.jar`. The Java EE specification defines JAR files as the packaging format for EJB components, Java EE clients, and Java libraries.

A stand-alone ejb-jar module contains EJBs and may be deployed directly to an application server or placed within an EAR. As illustrated in Figure 2-9, EJB JAR files contain Java class files and interfaces that are used to implement an EJB component, along with a deployment descriptor.

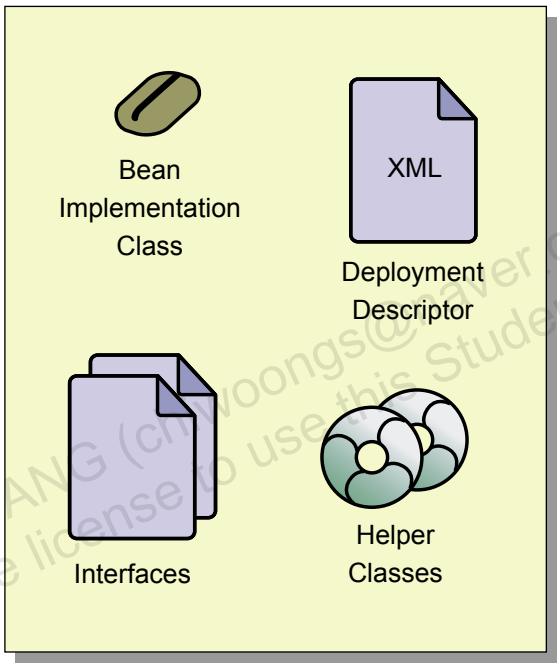


Figure 2-9 EJB Component JAR File Contents

Similarly, Java EE client components are packaged into client JAR files. The purpose of packaging client code is to allow the application server to generate a security infrastructure for the application. Remember that security is *not* the responsibility of the developer in the Java EE platform. With a web-based application, security is enforced at the web server level, so a client JAR file is not required. With a standalone Java application, the applications's security infrastructure limits the access that the client has to the EJB component tier in accordance with a security policy.

Resource Archive Files

A resource adapter is a software component that has hooks into a container's transaction management, security, and resource pooling subsystems. A resource adapter can request extended access to the system, beyond what would be allowed to an enterprise bean. Resource adapters can make native calls, create or open network sockets that listen, create and delete threads, and read and write files. None of these actions are allowed by enterprise beans on their own.

A component developer packages resource adapters into RAR files that have names that end in `.rar`.

RAR files might contain resource adaptors, Java classes, and a deployment descriptor.

Note – Resource adapters are beyond the scope of this course.

Enterprise Archive Files

The super archive is called an *enterprise archive* or *EAR file*. These files are usually given names that end in `.ear`.

EAR files are structured in the same way as Java Archive files. The EAR file contains EJB component archive files, web component archive files, resource adapter archive files, and client archive files.

Figure 2-10 illustrates the contents of an EAR file.

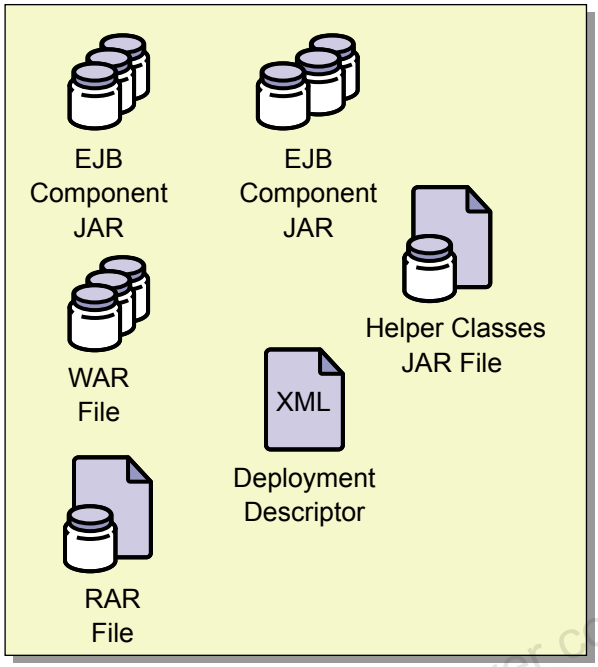


Figure 2-10 EAR File Contents

Deployment Descriptors

The Java EE specification defines a declarative way to describe the interactions between components and between components and their containers through the use of XML files called *deployment descriptors*. The deployment descriptor provides a means for configuring Java EE components and applications without requiring access to component source code.

Note – Deployment descriptors are optional beginning with Java EE 5. Developers can use in-code annotations to configure components. If present deployment descriptors override in-code annotations.

The deployment descriptor is an XML file. The file's format, naming convention, and location in the archive file is defined by the relevant component specification. For example, the EJB specification indicates that an EJB deployment descriptor must be a file called `ejb-jar.xml` that is located in a `META-INF` directory in the EJB JAR file or be located in the `WEB-INF` directory in a WAR.

A component or application deployment descriptor might specify:

- The transaction management strategy
- The authorization requirements, or who can do what to the component
- The deploy-time configuration variables
- The mapping to other components
- The external resource access dependencies

The Java EE platform stipulates that when components are assembled into an application, no changes to source code are allowed. The assembler might not even have the source code. Therefore, the use of the deployment descriptor is significant.

Vendor-Specific Deployment Information

Java EE component specifications might allow server vendors to include vendor-specific configuration files in the component archive files. For example, vendor-specific configuration files might contain information about the mapping of externally mappable names onto physical resources, load balancing properties, and session management settings. While the container might be able to supply defaults for some of these items, it cannot usually provide the externally mappable names for resources.

The implication of this is that the application deployer needs to provide the vendor-specific XML files, as required by the target platform. The vendor typically supplies a document type definition (DTD) or XML schema that describes the format and the data elements for the vendor specific configuration file. An application deployer can construct a valid vendor-specific configuration file using an XML editor and the DTD or XML scheme.

The effect of vendor-specific configuration on portability is minimal. These deployment activities are a small part of any project, and can easily be done for any server platform.

Summary

Component-based development is a key feature of the Java EE platform. The Java EE model defines a number of different component types, each tailored to fulfill a specific need. Java EE technology components have some or all of the following essential characteristics, depending on the component type:

- State and properties
- Encapsulation by a container
- Strict separation of interface from implementation
- Support for local and distributable component interactions
- Location transparency
- Component references obtained using a naming system

There are two basic ways that a component can interact with other components, synchronously or asynchronously. Although each style of interaction has its place in the Java EE programming model, the asynchronous model, in some cases, provides distinct advantages.

Java EE application development is typically performed by a group of people, each with separate roles and responsibilities. The Java EE specification defines several roles that are involved in the development, deployment, and management of an enterprise application. Developing a Java EE application typically involves several steps. An application developer uses a number of tools and techniques to develop a Java EE application.

Java EE application components are packaged into archive files for distribution and assembly into a Java EE application. Archive files contain a deployment descriptor that describes the characteristics of the components contained within an archive file.

Module 3

Web Component Model

Objectives

Upon completion of this module, you should be able to:

- Describe the role of web components in a Java EE application
- Define the HTTP request-response model
- Compare Java servlet components and JSP components
- Manage thread safety issues in web components
- Describe the basic session management strategies
- Describe the purpose of web-tier design patterns

Additional Resources

Additional resources – The following references provide additional information on the topics described in this module:

- Eric Jendrock, Debbie Carson, Ian Evans, Devika Gollapudi, Kim Haase, Chinmayee Srivathsa. “The Java EE 6 Tutorial,” [`http://java.sun.com/javaee/6/docs/tutorial/doc/`], accessed 1 August 2009.
- “RFC2616 Hypertext Transfer Protocol -- HTTP/1.1,” [`http://www.w3.org/Protocols/rfc2616/rfc2616.html`], accessed 1 August 2009.

Role of Web Components in a Java EE Application

Java EE applications often expose their functionality to browser-based clients using a suite of web-based components that are contained in a web container. Recall that there are two general Java EE application architectures that use web components, web-centric and EJB component-centric.

Web-Centric Java EE Application Architecture

Figure 3-1 illustrates the architecture of a web-centric Java EE application.

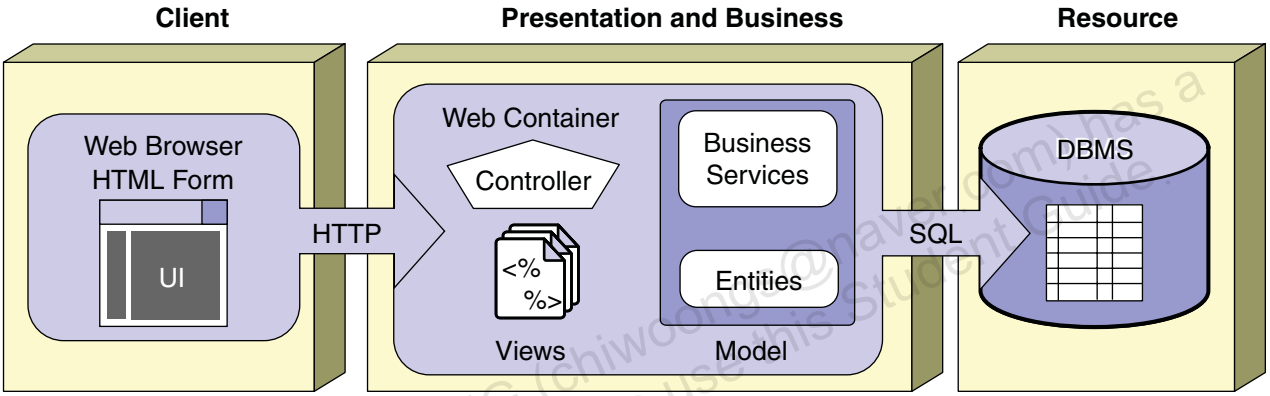


Figure 3-1 Web-Centric Java EE Application Architecture

In a web-centric application, components that reside in the web tier contain all of the presentation and business logic that is required by the application. A subset of EJB components can be used in this model with the new embedded EJB container, also known as EJB Lite.

EJB Component-Centric Java EE Application Architecture

Figure 3-2 illustrates the architecture of an EJB component-centric Java EE application.

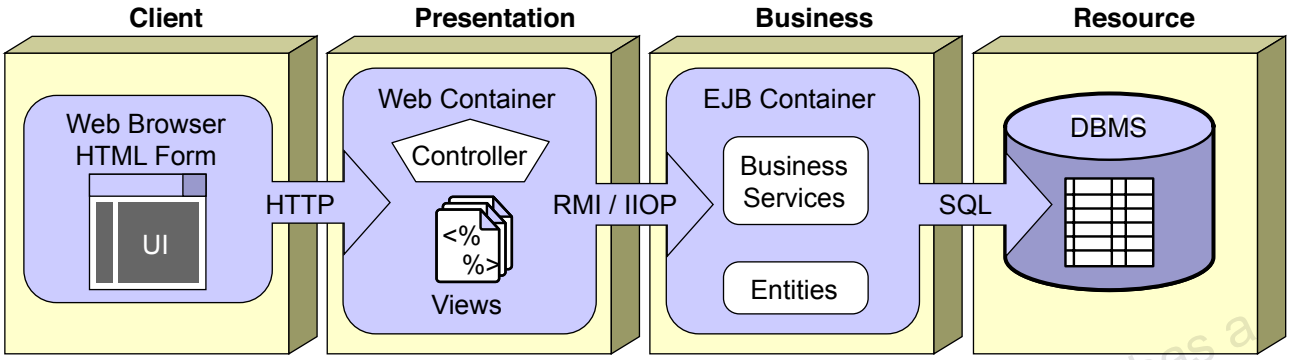


Figure 3-2 EJB Component-Centric Java EE Application Architecture

In an EJB component-centric application, the application implements the presentation mechanism using a suite of components that reside in the web tier. The EJB tier components implement the application business logic and associated data model. In an EJB component-centric application, web components perform a series of tasks that might include processing the incoming request, retrieving information from the request, invoking the requested business function, and generating a response that contains the results of the business process. Some of the components in the web tier might be responsible for request processing and flow control while other components might generate the response or view elements.

HTTP Request-Response Model

Browser-based clients access the functionality of Java EE web tier components using the Hyper Text Transport Protocol (HTTP). The HTTP is a protocol to transfer files between a server and a client. HTTP was created in conjunction with the related Hypertext Markup Language (HTML) standard.

HTTP defines a strict request-response sequence of interactions. Each request must be sufficiently complete for the application to carry out the requested operation, and each response must be sufficiently complete for the browser to display a page. No interaction between the browser and server is allowed, except for this sequence. Specifically, the server cannot send data to the browser, except when it is requested.

Note – HTTP was developed alongside HTML to support interaction between a web browser and a web server. HTTP is now increasingly used to carry data between applications in business-to-business applications and web services. In these applications, the data that is carried is less likely to be HTML, and the request data will not come from a browser. This does not change the basic technology, and the principles that underlie the development of web components remain the same. For the sake of brevity, in this module, it is assumed that the client of the web application is a web browser.

Figure 3-3 shows the HTTP request-response model.

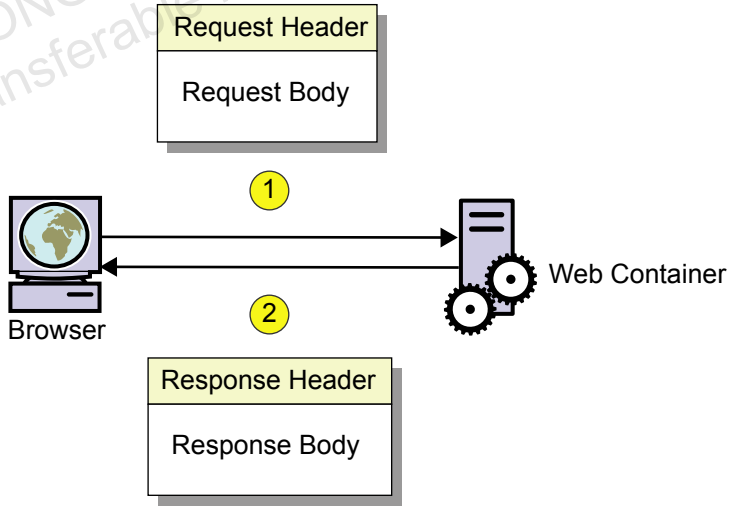


Figure 3-3 HTTP Request-Response Model

Each request consists of a header that contains the request type, the Uniform Resource Identifier (URI) of a particular resource, and a set of name-value pairs that provide further information about the request. This is followed by an optional request body.

The response consists of a header that contains the status code and a set of name-value pairs that provide further information about the response. The response header is followed by the response body. The body typically contains content to be rendered by the browser. However, if the client is not a browser, but another application, it might be more appropriate to supply data in a different format, often XML.

Despite its origin as a simple file transfer mechanism, HTTP has become a complex protocol, and a detailed presentation is beyond the scope of this course. A full description of the protocol is available at `http://www.w3.org/Protocols/rfc2616/rfc2616.html`.

Note – There is a technical distinction between a URI and a URL. A URL denotes the location of a file, that is, its path from the server's document root. URI is a more general term, and is any string that is interpreted by the server, which then locates the resource based on some mapping or search. Requests that are made for web components are more properly termed URIs because the correspondence between the URI string and the resource it identifies is an indirect one. The distinction is rarely important in practical development work.

The GET and POST Requests

The HTTP specification defines seven types of requests:

- GET
- POST
- PUT
- TRACE
- DELETE
- HEAD
- OPTIONS

In practice, most web applications use only the GET and POST requests.

The file upload functionality of the PUT request is now usually implemented by the POST request using MIME-encoded data. Browsers rarely issue the other request types. The GET and POST requests have similar capabilities, because they both supply a request that contains a URI, and they both expect some data to be returned in the response body. The differences between GET and POST requests can best be seen in the way that form data is sent from the browser to the server. When the HTML page contains a form that the user fills out, the values that the user enters must be transmitted to the server in the request. The form can be coded to instruct the browser to use either a GET request or a POST request.

The GET Request

The GET request type is the default. Regardless of whether the request is a form submission, a browser issues a GET request unless it is specifically instructed to do otherwise. In the GET request type, the browser appends any form data to the URI, and the request body is empty. Only a relatively small amount of form data (typically a few hundred bytes) can reliably be sent this way.

By convention, the browser displays the form data along with the URI, so the GET request should not be used to submit form data, which must not be exposed for viewing. For example, if the user submits a login form, the browser might display the form data with the URL, as shown in the following example:

```
http://www.mybank.com/bank/login?user=fred&pass=secret
```

The POST Request

The browser issues a POST request when it submits a form with a definition in HTML that contains the METHOD='POST' attribute as follows:

```
<FORM ACTION='form_test' METHOD='POST'>
```

The request body supplies the form data to the server. Modern browsers and servers use *chunked encoding* for POST request data, which means that the data is exchanged in blocks. Consequently, there is no limit to the amount of data that can be sent. In addition, the browser does not normally display the form data along with the URI, so the POST request is safer to use when the application sends confidential data that is entered by the user. In practice, it is nearly always preferable to specify the POST request rather than the GET request on HTML forms.

Form Data

When a web browser submits a form to the web server, the form data is passed in the HTTP request in an encoded format. Consider the form that is defined by the following snippet of HTML:

```
<FORM ACTION='form_test' METHOD='POST'>
<INPUT NAME='input1' SIZE='20'/>
<INPUT TYPE='SUBMIT' VALUE='OK'/>
</FORM>
```

The ACTION attribute defines the URI that is issued in the request, while the METHOD attribute defines the request type. The INPUT elements define the form input elements. In the previous code example, 'input1' is the only form input element. This form appears in the browser, as shown in Figure 3-4.

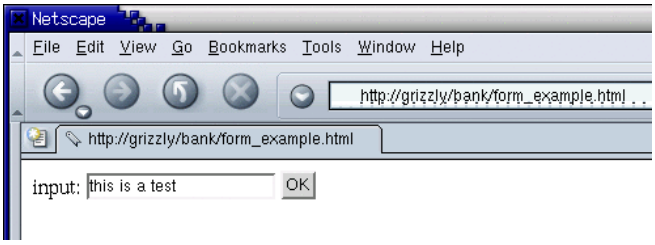


Figure 3-4 Form With One Input Element

When the user presses the submit button, the browser issues a request like the one in the following example:

```
POST /bank/form_test HTTP1.1
... request headers...

input1=this+is+a+test
```

The `/bank/form_test` URI is derived from the form's `ACTION` attribute.

Note – The URI given in the `ACTION` attribute is a relative URI, because `form_test` does not begin with a slash (/). However, the URI in the request header is `/bank/form_test`. The browser completes the expansion of `form_test` to `/bank/form_test`, because the URL of the form itself begins with `/bank`. If the `ACTION` attribute had specified `/bank/form_test`, exactly the same request would have been issued, but the presence of an absolute URI in the HTML form makes it more difficult to maintain. As a general principle, the best attribute to use is the shortest `ACTION` attribute that generates the correct request URI.

In a `POST` request, the form data is sent by the browser in the HTTP request body, which follows the request headers. The form data is encoded so that spaces are rendered as plus (+) signs. Consequently, the text `this is a test` is encoded as `this+is+a+test`. Other non-alphanumeric characters are converted into hexadecimal. The web container decodes the form data before passing it to the web component.

Content Type and the Response Header

The response that is returned by the server provides information about the data that is supplied in the body. Specifically, the `content-type` header defines the data type that follows in the response body. When the client is a web browser, the data is likely to be either HTML or some other type of data that can be rendered for display, as shown in the following examples:

- An HTML page – `text/html`
- An XML document – `text/xml`
- An image in JPEG format – `image/jpeg`

The response header can also indicate the length of the response body and the character encoding that it employs.

Comparison of Servlets and JSP™ Components

There are two basic types of web components that are used in a Java EE application, servlets and JavaServer Pages technology components. The web tier in a Java EE application might also contain static elements, such as HTML pages or image files, and other helper and utility classes.

Servlets are web components that are authored in the Java programming language. In principle, a servlet is any class that implements the `Servlet` interface. In practice, developers take advantage of the base classes that are provided by the Java EE platform's API. Typically, a servlet implementation extends the `HttpServlet` class. This class provides boilerplate handlers for the HTTP request types that are not of interest in a web application, such as `DELETE` and `OPTIONS`. The `service` method in the `HttpServlet` class delegates to methods called `doGet`, `doPost`, and so on. You can override these methods as required. The `doGet` and `doPost` methods typically carry out the work of the servlet.

JSP components consist of presentation content (typically HTML) with embedded programmatic elements. JSP technology is designed to be manageable by content authors, rather than by developers. In principle, it is permissible to embed Java programming language operations in a JSP component. However, in practice, a better strategy is to implement custom functionality in custom tags.

In practice, JSP components are translated into servlets by the web container. This translation happens at some point before the first request is delivered.

Web Component Management and Life Cycle

Servlets and JSP components (post-translation) have the same life cycle, and are managed by the web container in the same way. The life cycle is summarized in the sequence diagram of Figure 3-5.

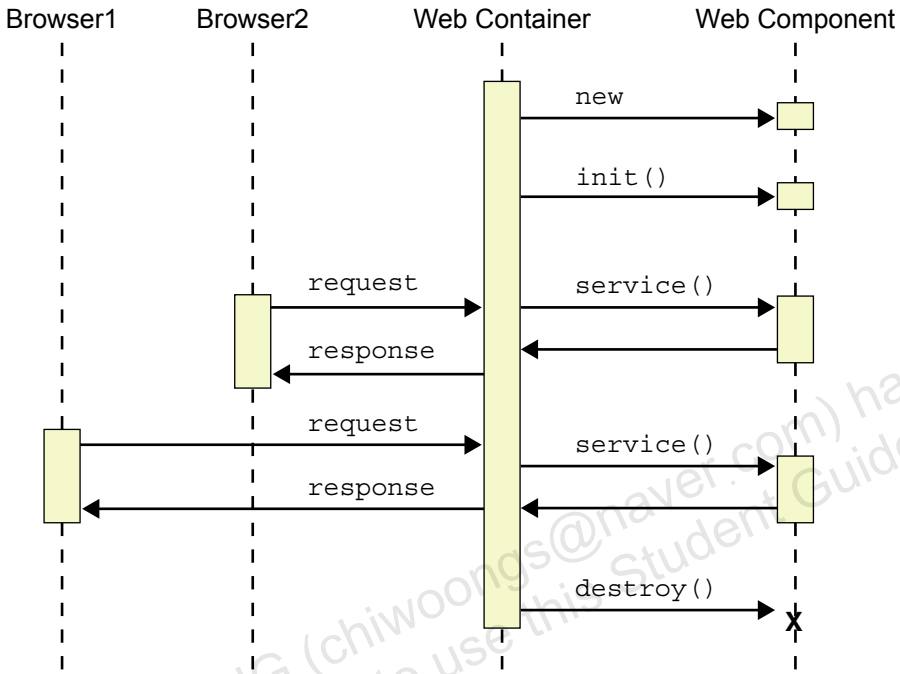


Figure 3-5 Life Cycle of a Web Component

The web container initializes the web component at some point before the first request, and calls its `init` method. The `init` method is guaranteed to be called exactly once. Thereafter, each request from the browser results in a call on the `service` method. The container passes two arguments to the `service` method, one argument represents the HTTP request and the other argument represents the HTTP response. The web container can remove an instance of a web component. If the web container removes an instance of a web component, it first calls the `destroy` method.

Note – Java EE 6 web components can provide additional life-cycle methods using annotations. This is covered in Module 4, Developing Servlets.

The `service` Method

The web container calls the `service` method once for each incoming request. Developers do not typically supply the service method, instead the service method calls additional developer supplied code. The execution of the `service` method typically completes the following operations:

- Validates any form data
- Updates the application's data model
- Collects data from the model to be rendered
- Renders the data in HTML *or* passes the request and the data to another component to be rendered

Not all components carry out all of these operations on every request. Specifically, it is now uncommon to carry out form processing in JSP components, or to carry out HTML rendering in servlets.

Servlet and JSP Component Examples

This section gives basic examples of servlets and JSP components. At this stage, you are not expected to understand these examples in detail. Rather, the examples are intended to illustrate the similarities and differences in the technologies.

Servlet Example

The code snippet in Code 3-1 shows a trivial, but complete, servlet implementation.

Code 3-1 Example Servlet Implementation

```
1   package com.example;
2
3   import java.io.*;
4   import java.util.Date;
5   import javax.servlet.annotation.WebServlet;
6   import javax.servlet.http.*;
7
8   @WebServlet("/hello")
9   public class HelloServlet extends HttpServlet {
10
11      @Override
12      protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws IOException {
13          response.setContentType("text/html");
14          PrintWriter out = response.getWriter();
15          out.println("<html><head/><body>");
16          out.println("<h1>Hello, World!</h1>");
17          out.println("The date is:" + new Date());
18          out.println("</body></html>");
19          out.close();
20      }
21  }
```

The `doGet` method, which is executed by the service method in this example, takes two parameters. The parameters are supplied from the container:

- The `request` parameter provides information about the request that is supplied by the browser. Its values are essentially object representations of the data that is supplied by the browser in the HTTP request.
- The `response` parameter allows the servlet to generate output.

The servlet must generate the output exactly as it is to be seen by the browser. In the previous code example, the servlet generates HTML. The purpose of this example is to show the general approach to implementing a servlet.

JSP Component Example

Output similar to what was generated by the preceding servlet example can be produced using a JSP component, as shown in Code 3-2.

Code 3-2 Example JSP Component Implementation

```
1   <%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>
2   <html>
3   <head/>
4   <body>
5   <h1>Hello, World!</h1>
6   <jsp:useBean id="date" class="java.util.Date" />
7   The date is: <fmt:formatDate value="${date}" type="both" />
8   </body>
9   </html>
```

The most obvious difference between the representation of the servlet and JSP component is that the JSP component does not need programmatic constructs to generate output to the browser. In a JSP component, all of the content is output except the content that falls between special symbols and xml tags. In Code 3-2, the *declaration* `<%@ taglib ... %>` is used. A `<jsp:useBean>` tag is used to create a `Date` object. The JSTL `<fmt:formatDate>` tag renders the Date object as a formatted string.

Servlet and JSP Component Collaboration

Servlets and JSP components have essentially the same capabilities, but they are expressed differently. In general, JSP components are useful for generating presentation, particularly presentation in HTML and XML. On the other hand, servlets are useful for processing form data, for computation, and for collecting data for rendering. Therefore, most modern web applications use servlets and JSP components in collaboration. Typically, requests are handled by servlets, which do the computation and then transfer control to JSP components, which complete the rendering. This collaboration allows the benefits of each type of component to be exploited.

A `RequestDispatcher` object effects the transfer of control between web components. A web component can use either the `forward` or `include` method of the `RequestDispatcher` to transfer control. The `forward` method transfers control to a target component entirely, while the `include` method merges the output of the target component into the output of the calling component.

Run-Time Behavior of Servlets and JSP Components

Because JSP components are translated into servlets, JSP components are essentially servlets at run time, and have the same properties as servlets. Specifically, both servlets and JSP components have the same life-cycle and container management, the same API, and the same access to the client session.

Of particular concern to you is the fact that both servlets and JSP components can be entered on multiple threads and must be implemented accordingly. This applies also to tag libraries that are developed for inclusion in JSP components.

Managing Thread Safety Issues in Web Components

In a busy system, a web application might be simultaneously handling requests from many browsers. In the early days of web applications, it was usual to invoke an operating system process to handle each request. While this had the advantage of keeping the requests well separated, it did not make good use of centralized processing unit or memory resources. The web component model of the Java EE platform takes the approach of assigning requests to threads, not to processes. To obtain a good average throughput, a web server usually maintains a pool of threads, and assigns each incoming request to a thread as soon as one becomes free. Whenever a thread finishes its work in a particular servlet, it is reassigned to the free pool. The server might also maintain a pool of instances of each web component to simplify thread management, although this is unlikely for performance reasons. However, no guarantees are offered about which threads are assigned to which instances.

This section describes the implications that these design decisions have for the web component developer.

Web Component Thread Model

The web component developer should assume that any number of threads can be executing in the service method at any particular time. Figure 3-6 illustrates one of the many situations that can be problematic in such a multi-threaded environment.

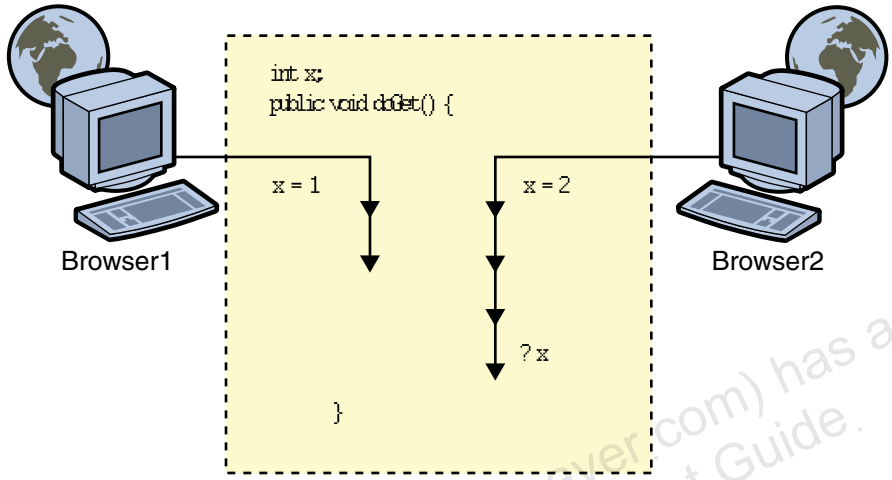


Figure 3-6 Web Component Thread Model

In this example, browser 2 issues a request that is handed to the `doGet` method. The method performs a calculation and updates the value of instance variable *x*. While browser 2's thread is still active, browser 1 issues a request that causes the `doGet` method to be entered again. It performs a calculation, and updates the value of *x*. Because *x* is an instance variable, and instance variables are shared by all threads, this updates the value seen by browser 2's thread.

It might be the case that this potential problem is overcome simply by declaring *x* to be a variable local to the `doGet` method, but not all thread-related problems are so easy to handle.

Implications for the Developer

Because the web container might allow a web component's service method to be entered on multiple concurrent threads, web components must be developed to be thread-safe. While it is impossible to give specific information about how this is to be accomplished, experience and thorough testing will reveal whether there is likely to be a problem in a specific implementation. However, there are a few basic principles that you should observe.

- Use instance variables cautiously

  A variable should be assigned local scope if this does not compromise the application logic. Instance variables can safely be used if they are *final*, or if they are assigned during the initialization of the web component and remain constant thereafter.

- Use class variables cautiously

  The same problems can arise with class variables as with instance variables, but even more acutely. The added complication is that the use of the `synchronized` construct does not necessarily prevent class variables from being written and read by different threads concurrently. The web container is not forced to deploy exactly one instance of the component in a distributed application. The web container might have multiple instances in multiple virtual machines. Many developers prefer to avoid the use of class variables altogether, except those that are declared *final*.

- Provide access to external resources cautiously

  Concurrent access to relational databases is rarely a problem, because the database engine has its own locking mechanism. However, web components that read and write files, as well as update legacy systems, might have to be more careful.

- Use synchronization constructs to denote critical sections

  Do not make the scope of the critical section any larger than it needs to be. The following code snippet creates a block of code that can be entered only on one thread at a time:

  ```
  synchronized (this) {
  // This section is only entered by
  // one thread at a time
  // ...
  }
  ```

You can also use the functionality in the `java.util.concurrent` package to create thread safe code without the overhead of applying the synchronization constructs.

Web Context Root and Alias Mapping

Servlets and JSP components are packaged into a web application, along with any static content, such as HTML or images, that is required. The application is typically deployed in a Java archive that is conventionally called a WAR file. As part of a complete application, the WAR file is, in turn, packaged into an EAR file.

When it is deployed this way, the web application is assigned a *context root*. The context root identifies a specific web application on the server, which might be hosting multiple applications. Therefore, the URI that is issued by the browser must include the context root, in addition to any identifier that is required to reference a specific component. The URI has the following form:

```
http://server:port/context_root/resource
```

The resource field can reference a specific file, or it can be an alias for a servlet. If the resource field references a specific file, the resource identifier is a path that is relative to the root of the web application. If the resource identifies a servlet, it does so by means of an alias that is created in the servlet class with an annotation or the web application's deployment descriptor. For example, in the following URI, `bank` is the context root, while `main` is the alias of the servlet:

```
http://www.mybank.com/bank/main
```

The `main` alias is mapped to the Java class that implements the servlet.

Session Management

Because HTTP is stateless, the server cannot ordinarily distinguish between two successive requests from the same browser and a single request from each of two different browsers. This has traditionally made it difficult to develop web applications that employ extended *conversations* between the browser and the application. The Java EE platform provides a simple session management strategy in which each browser is allocated a private session object on the server.

Session Management Strategies

HTTP is stateless, so a conversational session cannot be managed by the protocol itself. Therefore, the management of the session falls on the component developer, at least to a degree. Session data can be stored either on the browser or on the server. Although session data can be stored on the browser, it cannot be *processed* on the browser. So, session data must be submitted to the server on every request and then returned by the server to the browser in modified form.

Session Management Techniques

There are various techniques that are available for storing session data on the browser so that it can be sent to the server and returned on every request. Cookies are the most effective technique for storing session data on the browser, but not all browsers support cookies. Even where cookie support is available, it can be inefficient to shuttle large volumes of session data back and forth between the browser and the server on every request. When the volume of the session data is small, storage on the browser can be effective because it is simple and has little impact on server resources.

In practice, most applications make use of session storage on the server. This practice allows better use of the available network capacity, and imposes less stringent limits on the amount of session data. However, it is still necessary to store some information on the browser. At a minimum, this data includes a session identifier to allow the server to identify the client and to select the correct session data. Java EE session management strategies are presented in detail in the next module.

Managing Complexity in the Web Tier

Web-based user interfaces rapidly become complex interfaces to develop and manage. This section explains why this is so, and outlines some of the general strategies that you might adopt to manage the complexity.

Problems With Web-Tier Development

A number of factors come together to make development and management of web-based user interfaces more complex than it initially appears:

- The application must address the fact that HTTP follows a strict request-response sequence and is stateless. The application's business logic cannot notify the user interface that the underlying data has changed, so each request from the browser must reconcile the user interface's view of the state of the application with the business logic's view.
- It is easy to end up with large numbers of servlets and JSP components that all handle different types of requests. For example, servlet 1 might generate an HTML page which contains a form that activates servlet 2. Then servlet 2 generates an HTML page that reactivates servlet 1, and so on. This *spaghetti* of dependencies between web components is difficult to track in a large project.
- JSP components are good for presentation, but lose their benefits when they contain embedded Java programming language statements. At the same time, servlets are not ideal for generating presentation.

A well-designed application takes advantage of the request-response interaction, and uses it to structure application flow, rather than seeing the request-response interaction as a limitation. For example, you can view the application flow in terms of a finite-state model. The requests from the web browser represent events in this model, while the actions that are carried out by servlets represent the transitions between states. Because you cannot change the way that the browser and the server interact, you should try to use it to good advantage.

Model 1 and Model 2 Architectures

Two broad design strategies have emerged to handle the use of JSP components in the web tier, which have become known as *Model 1* and *Model 2* architectures. You should be aware that the terms Model 1 and Model 2 architectures are jargon terms that are subject to slightly different interpretations because they are not defined in any specification. In general, the following essential characteristics apply to the Model 1 approach:

- JSP components handle request processing through `<jsp:useBean>` classes, the standard tag library, and custom tags discussed in Module 5.
- JSP components render data that is retrieved from the business-logic tier.

The Model 2 approach, however, uses servlets and JSP components in collaboration:

- Servlets handle request processing.
- Servlets interact with the business logic and collect data for display.
- JSP components render the data for display.

The Model 2 architecture can be seen as an implementation of the Model-View-Controller design paradigm that is described in the next section.

Model-View-Controller Paradigm

The Model-View-Controller (MVC) paradigm in its traditional sense has a long history. It first came to prominence in the early 1980s as a means for structuring SmallTalk applications. Figure 3-7 shows the architecture of the traditional MVC paradigm.

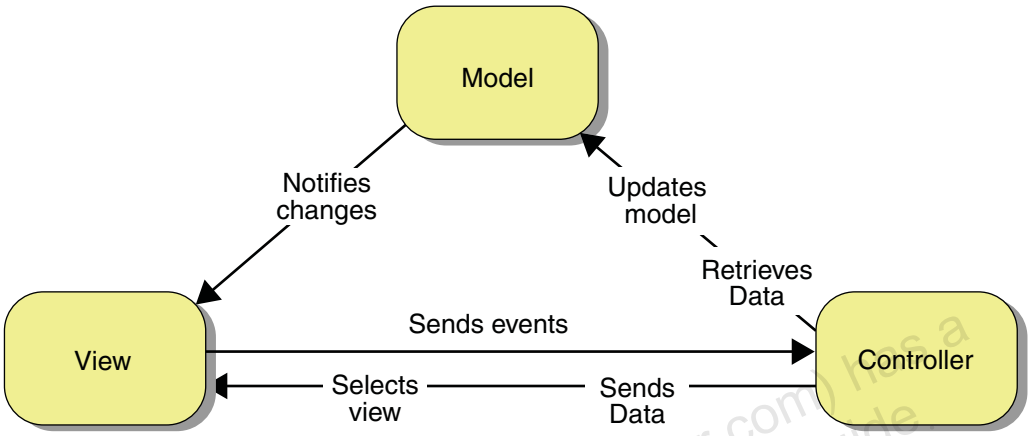


Figure 3-7 Traditional MVC Architecture

The essential feature of the MVC paradigm is that it divides the application logic into three distinct roles that have well-defined responsibilities towards each other.

- The *model* element includes those parts of the application that represent and manage the underlying data and state of the application.
- The *view* element includes application components that render the model on behalf of the user. That is, the view element contains the parts of the application that directly face the user. The view should not modify the model, but only render it. The view can, however, pass user interface gestures, such as button presses, mouse clicks, menu selections, and so on, to the controller element
- The *controller* element updates the model in response to user interface gestures. It also selects the view based on user gesture or model logic.

Model 2 Architecture as a Realization of the MVC Paradigm

The Model 2 architecture can be mapped onto the MVC paradigm, as shown in Figure 3-8.

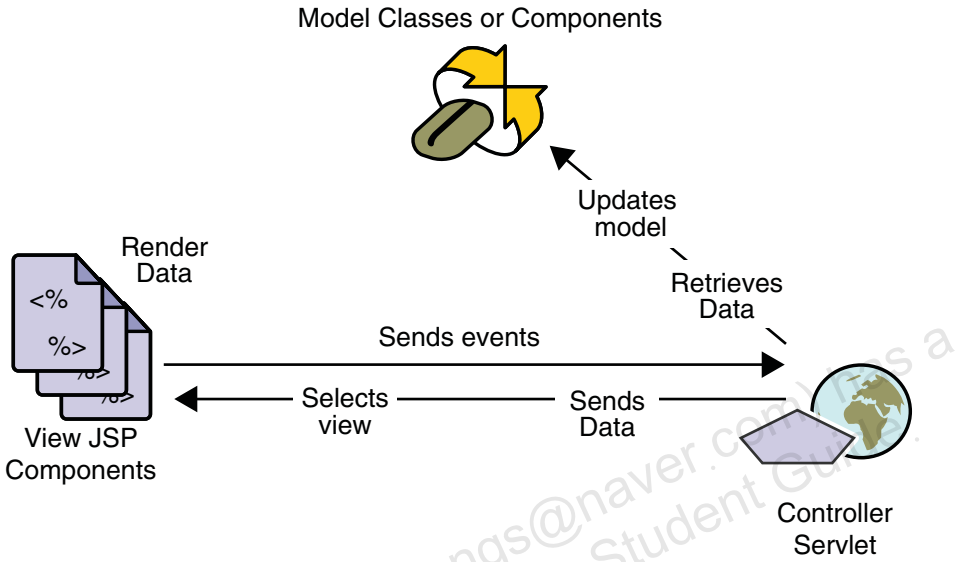


Figure 3-8 Model 2 Architecture as MVC

In this mapping, the view elements are usually JSP components. These components are well-suited to rendering data, but are not suited for processing data. The role of the JSP components as view elements means that they are not required to do any processing, so this limitation is avoided.

The controller elements are typically servlets. Servlets are not ideal for presentation, but their role as controllers mean that they are not required to do any presentation.

Model classes may be implemented as Plain Old Java Objects (POJOs), as EJB components, or a mixture of POJO and EJB components. The composition of the model depends on the type and scale of the application that is being developed.

MVC in the Java EE Platform

In the Java EE platform the model, view, and controller responsibilities are typically allocated to components as follows:

- Controller – Usually a servlet. Typically the servlet delegates to helper classes to carry out specific actions, rather than addressing them internally.
- View – Usually one or more JSP components, but also HTML files and other static content. Some applications use the XML Stylesheet Language for Transformations (XSLT) as well.
- Model – When the application includes EJB components, the model is usually either a facade session bean, or a Business Delegate object that represents a facade session bean.

  When there are no EJB components, the model is usually comprised of classes that encapsulate the business services.

Using Web-Tier Design Patterns

Model-View-Controller and Model 2 architecture are broad, architectural paradigms. They do not describe, nor even guide, the development process for web-tier applications. A number of design patterns have emerged that realize these architectural paradigms in a way that can be used to guide implementation.

This module considers three specific patterns, which can be used in combination:

- Service-to-Worker
- Dispatcher View

  The Dispatcher View pattern is similar to the Service-to-Worker pattern.
- Business Delegate

There are many other patterns that are documented in the standard patterns catalogs. In addition, standard pattern catalogs often present a number of different versions of the Service-to-Worker and Dispatcher View patterns, each with its own particular benefits.

Note – A detailed description of web-tier design patterns is outside of the scope of this course. If you are interested in this particular topic, it is covered in more detail in SL-500, *Java EE Patterns*.

Service-to-Worker and Dispatcher View Patterns

The Service-to-Worker and Dispatcher View patterns are both realizations of a Model-View-Controller implementation. Both are widely used, and they differ only in the way in which the *view* elements (JSP components) interact with the model elements.

On each request the web-tier application performs a sequence of steps as shown in Figure 3-9.

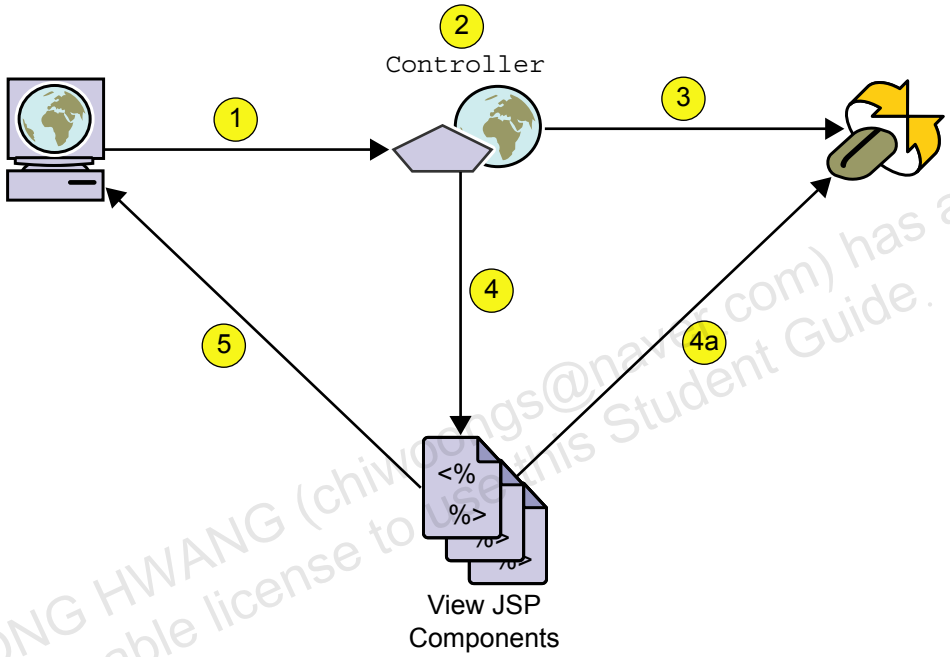


Figure 3-9 Service-to-Worker and Dispatcher View Patterns

The steps that are completed by the web-tier application are as follows:

1. The web browser issues a request. The request might be part of an ongoing conversation, so there might be state associated with it.
2. The `Controller` servlet examines the request and the state of the application and validates any form data.
3. The `Controller` servlet carries out the necessary operations on the model elements. In the Service-to-Worker pattern, the controller also collects from the model, whatever data should be rendered for the display.

4. The `Controller` servlet selects the view element that is needed to render the data and dispatches to that view element.

   With the Dispatcher View pattern, the view element can collect data from the model elements as required. However, the views are forbidden from modifying the model. In the Service-to-Worker pattern, the servlet passes all of the required data to the view element.

5. The view elements render the data into the format required by the browser, usually HTML. The view elements might use custom tags when the rendering is non-trivial.

Web-Tier Design Framework Construction

With the Service-to-Worker and Dispatcher View patterns, each request is handled in the same generic way, with variations for the specific needs of the application. All applications require some generic capabilities, such as a method to control flow through the application, a process to validate form data, a form of interaction with the data model, and so on.

This generic functionality can be extracted from the application to form a design framework that can be used in other applications. The more generic and general-purpose the framework, the easier it is to reuse.

In practice, the controller element is the most generic, and the same controller can be used in many different applications. Because the controller's role is so prescriptive, it can often be expressed declaratively, rather than in code.

Some Available Web-Tier Frameworks

The web-tier developer can construct a framework to suit each individual application, or to attempt to separate the *framework* elements from the *application* elements of the design. The framework elements might then be reusable in future projects. However, this might prove to be unnecessary. You should use a framework that meets your needs, but unless you have a lot of time and money, you are usually better off to use one of the general-purpose frameworks that are already available. This section describes a few of the better-known general-purpose frameworks:

- Struts – One of the earliest and most widely-used web frameworks

  The Struts framework is well-suited to projects of low to moderate complexity. It includes a controller element whose behavior can be specified declaratively in XML or by annotations placed in action classes. Struts also includes a set of custom tags to format the output of business logic elements.

- JavaServer™ Faces (JSF) technology – JavaServer Faces technology simplifies building user interfaces for JavaServer applications. Developers of various skill levels can quickly build web applications by: assembling reusable UI components in a page; connecting these components to an application data source; and wiring client-generated events to server-side event handlers. All Java EE web containers are required to support applications that use the JavaServer Faces technology.

Web-Tier Decoupled From the Business Logic

Another widely-used web-tier design pattern is the Business Delegate, which is illustrated in Figure 3-10.

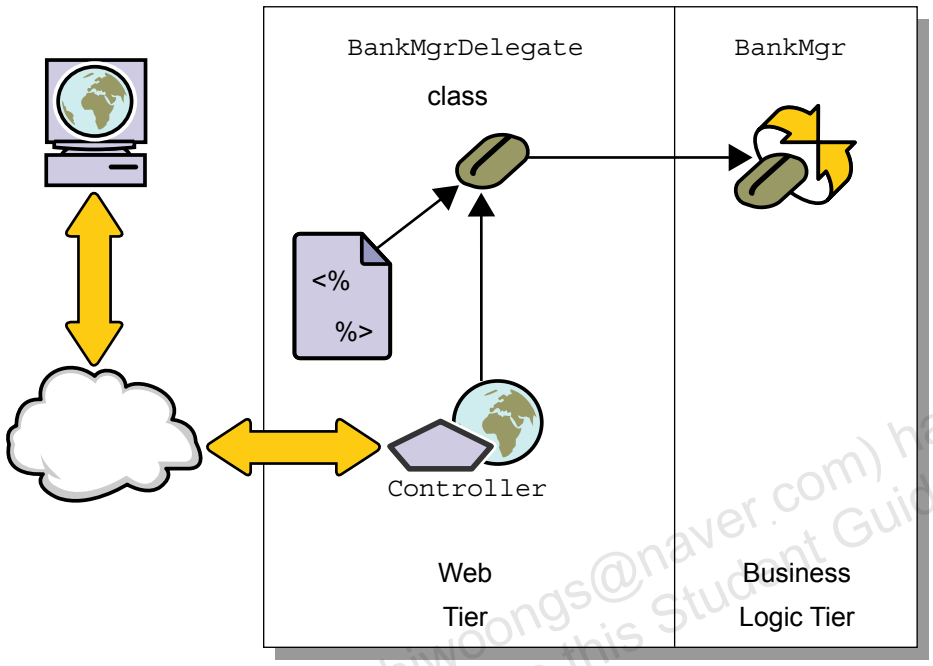


Figure 3-10 Business Delegate Pattern

You can implement a servlet that makes method calls directly on EJB components. To do so, the servlet must look up and cast the EJB and store the reference in a suitable location or use dependency injection. The servlet must also handle exceptions that are thrown from the component.

None of these operations is particularly difficult, but there is a case for saying that a servlet should not need to be developed with a knowledge of EJB technology. A Business Delegate pattern is a representation in the web tier of the services that are provided by an EJB component. It encapsulates the operations that are specific to EJB components, such as exception handling and JNDI API calls, and presents a simplified interface to the servlet.

Summary

The Java EE platform supports two web-tier components, servlets and JSP components, that are variants of one another. They have the same run-time characteristics, and have access to the same APIs. Servlets are better suited to expressing presentation logic, while JSP components suit the delivery of presentation content. Both component types have access to the platform's session management infrastructure.

The developer of a Java EE application must balance a number of conflicting goals. For example, while a strict object-oriented approach to software design might be desirable, to increase reuse and reduce software maintenance costs, performance considerations dictate that the application should minimize the total number of method calls.

This restriction on method calls inevitably fetters the developer's freedom to use a strict object-oriented design. These trade-offs have been extensively investigated by Java EE application developers, and the result is a growing set of design patterns that have been found to give a reasonable compromise between the conflicting design requirements.

Module 4

Developing Servlets

Objectives

Upon completion of this module, you should be able to:

- Describe the servlet API
- Use the request and response APIs
- Forward control and pass data
- Use the session management API

Additional Resources

Additional resources – The following references provide additional information on the topics described in this module:

- Eric Jendrock, Debbie Carson, Ian Evans, Devika Gollapudi, Kim Haase, Chinmayee Srivathsa. “The Java EE 6 Tutorial,” [`http://java.sun.com/javaee/6/docs/tutorial/doc/`], accessed 1 August 2009.
- "JSR 315: Java Servlet Specification, Version 3.0", [`http://www.jcp.org/en/jsr/detail?id=315`], accessed 1 August 2009.

Basics of the Servlet API

The servlet API provides the following facilities to servlets:

- Callback methods for initialization and request processing
- Methods by which the servlet can get configuration and environment information
- Access to protocol-specific resources, such as the client's session object

This section presents a brief description of the classes and interfaces that make up the servlet API.

Generic and Protocol-Specific APIs

The servlet API is divided into three levels, as shown in Figure 4-1.

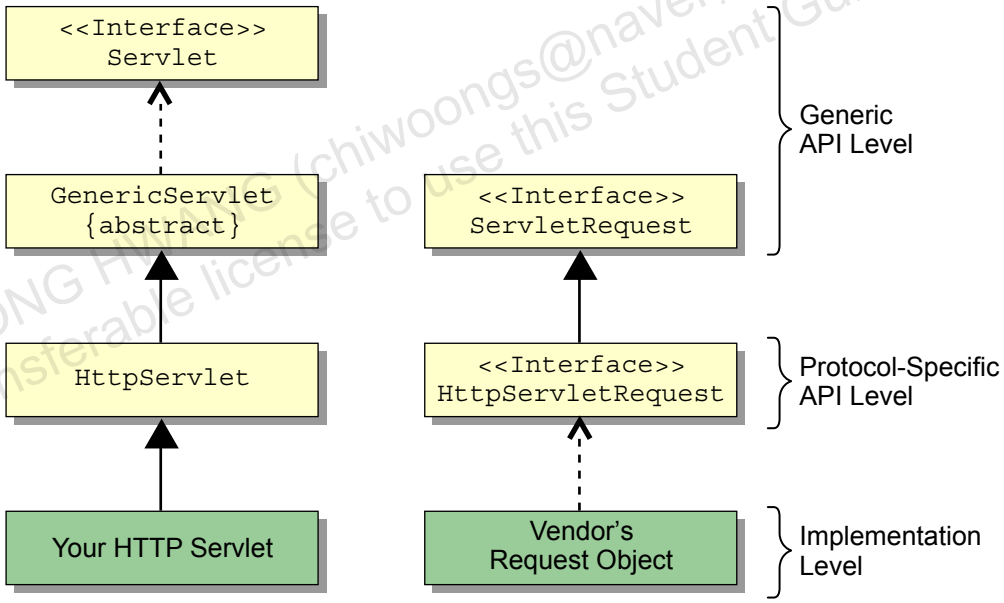


Figure 4-1 Structure of the Servlet API

The three levels of the servlet API can be described as follows:

- The top level consists of interfaces and abstract base classes that are not specific to any particular communications protocol.
- The second level consists of abstract classes and subinterfaces that extend the generic functionality for specific protocols.

  At present, the servlet API is mostly used with servers that communicate with clients using HTTP. However, there is no reason why other protocols should not be supported. For example, with the appropriate server-side infrastructure, it would be possible to implement servlets that respond to FTP requests.
- The third level consists of classes that implement the full capability of the layer above.

  These classes are either authored by the web component developer or they are part of the vendor's server implementation.

Benefits of the Protocol-Specific API

Any class that properly implements the `Servlet` interface or extends the `GenericServlet` class can be a servlet. However, it is usually more convenient to work with the protocol-specific classes and interfaces, such as `HttpServlet`, for a number of reasons:

- Protocol-specific classes and interfaces provide access to objects that are protocol-specific, such as the `HttpSession` implementation.
- The method arguments and return values are defined in terms of other protocol-specific objects.
- Protocol-specific classes and interfaces provide boilerplate processing for common operations, so that the application developer does not have to implement methods that are not relevant to the application.

When you implement a servlet that supports the HTTP operations, you typically work with the `HttpServlet` base class, the `HttpServletRequest` and `HttpServletResponse` interfaces, and the `HttpSession` session object. If the application server supports other protocols, it is expected that the application server will provide similar classes and interfaces for those protocols. At present, however, the Java EE specification does not mandate the support for protocols other than HTTP.

Benefits of the `HttpServlet` Class

Extending the `HttpServlet` base class, rather than implementing the `Servlet` interface directly, provides you with the following benefits:

- A simplified, no-argument `init` method that can be overridden to do initialization without the need to initialize the base class

  The servlet interface specifies a number of methods that a servlet can call, such as `getServletContext`. The class that implements the `Servlet` interface is expected to be able to initialize itself from the arguments that are supplied to the `init` method, which is defined in that interface. That initialization should do whatever is necessary to ensure that methods, such as `getServletContext` return meaningful values. The `HttpServlet` base class initializes itself and then calls `init` so that the specific servlet can initialize itself. This method helps you to avoid becoming involved in initializing classes that are part of the API and not the application.

- Standard handling of `HTTP` request types that are not of interest to the servlet

  For example, the servlet should handle requests of type `DELETE`, `TRACE`, and so on, *elegantly*. That is, the handling of the request should not expose a security weakness in the application or inconvenience the end user more than necessary. However, these will rarely be of interest to a web application. The `HttpServlet` class provides safe handling of these requests, typically by returning a *not support* message to the client.

- Request handler arguments that are defined in terms of HTTP-specific request and response objects

The `service` Method

The `HttpServlet` class provides an implementation of the `service` method that filters incoming requests by type. The `service` method also delegates control to a handler for the request type, as shown in Figure 4-2.

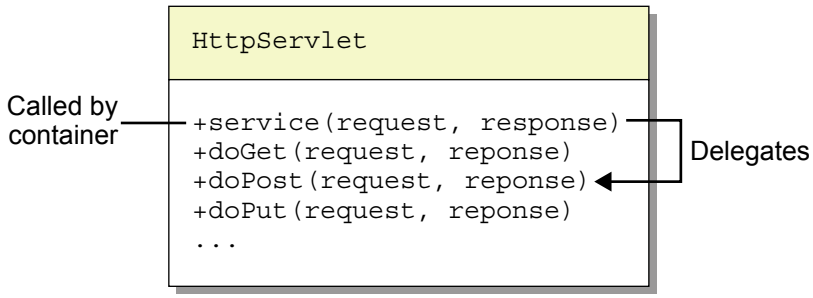


Figure 4-2 The `service` Method

For example, `GET` requests result in a call to the `doGet` method, `POST` requests result in a call to the `doPost` method, and so on. A servlet that is implemented as a subclass of `HttpServlet` should not override the `service` method, unless it is prepared to handle all of the `HTTP` request types, even those that a browser never sends. An unauthorized person who is trying to find a security vulnerability in your application will not feel constrained to use a web browser, so you do not want to leave this door open.

Request Handling Methods

In most cases, a web application is interested in handling GET and POST requests. A browser issues a GET request unless it is specifically told otherwise, so you should regard the GET request as the default. The browser uses a POST request to submit a form that is marked in the HTML to use this request type. The other request types are not usually of interest in a web application.

In practice, the servlet is not usually concerned about what type of request caused it to be invoked. The same kind of information is available to it in either case, and the servlet can make exactly the same API calls to handle the request. Therefore a common strategy is to make the goGet and doPost methods both call a common handler, which is typically called processRequest. This strategy is illustrated in Figure 4-3.

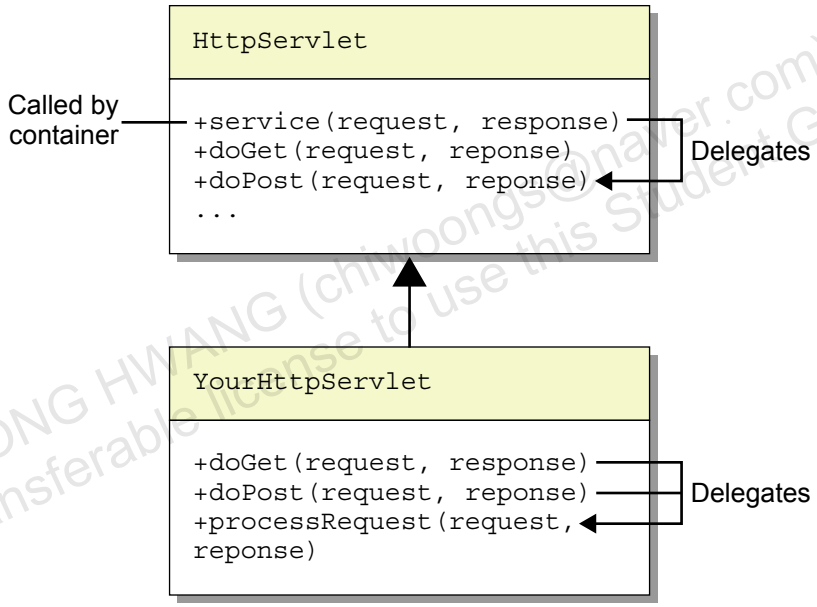


Figure 4-3 Providing Common Handling for GET and POST Requests

Note – In any given situation, it is important to correctly specify whether the browser makes a GET or a POST request. There are costs and benefits of these two types of requests. However, after the browser has made a request of a particular type, the servlet itself does not usually care whether it was a GET or a POST request. It will not always be appropriate to handle GET and POST identically, but it so frequently is, that the code examples in this module all assume that you have adopted this approach.

Code 4-1 shows a servlet that has been coded according to these principles for the common handling of `GET` and `POST` requests.

Code 4-1 Basic Servlet

```
1   package com.example;
2
3   import javax.servlet.annotation.WebServlet;
4   import javax.servlet.http.*;
5
6   @WebServlet("/example")
7   public class ExampleServlet extends HttpServlet {
8
9       @Override
10      public void doGet(HttpServletRequest request, HttpServletResponse
response) {
11          processRequest(request, response);
12      }
13
14      @Override
15      public void doPost(HttpServletRequest request,
HttpServletResponse response) {
16          processRequest(request, response);
17      }
18
19      public void processRequest(HttpServletRequest request,
HttpServletResponse response) {
20          // Process request and generate response
21      }
22  }
```

Servlet Configuration

Without configuration, a servlet does not have an accessible URL. Beginning with Servlet spec 3.0 (Java EE 6) annotations are used to map URLs to servlets. The following examples show the use of the `javax.servlet.annotation.WebServlet` annotation to configure URL mapping for servlets.

Code 4-2 Example of a single URL for a servlet

```
@WebServlet("/myservlet")
public class MyHttpServlet extends HttpServlet{
//...
}
```

Code 4-3 Example of multiple URLs for a single servlet

```
@WebServlet(name="SomeName", urlPatterns={"/myservlet",
"/foo", "/bar"})
public class MyHttpServlet extends HttpServlet{
//...
}
```

Deployment Descriptors

Servlets can be configured through an XML configuration file known as a deployment descriptor. Servlets can also be configured using Java annotations; however, annotations are optional and can be disabled.

The servlet deployment descriptor must be named `web.xml` and placed in the `WEB-INF` directory of the web application. Any configuration details placed in the `web.xml` file take precedence over any annotation based configuration.

Code 4-4 Disabling Annotations in the Deployment Descriptor

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
version="3.0" metadata-complete="true">
</web-app>
```

Note – The `web.xml` file has been optional since Servlet specification 2.5 maintenance review 2. However, in that version the deployment descriptor was only optional if the application did not contain Servlet, Filter, or Listener components. Servlet specification 3.0 (Java EE 6) makes the deployment descriptor truly optional with the addition of annotations for almost all aspects of web application configuration.

To configure a servlet and many other aspects of a web application with a deployment descriptor, place a `web.xml` file in the web applications `WEB-INF` directory.

Code 4-5 A Basic `web.xml` Deployment Descriptor

```
<?xml version="1.0" encoding="UTF-8"?>

<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">

    <servlet>
        <servlet-name>SomeName</servlet-name>
        <servlet-class>package.MyHttpServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>SomeName</servlet-name>
        <url-pattern>/myservlet</url-pattern>
    </servlet-mapping>

</web-app>
```

Servlet Life Cycle

As outlined in the previous module, servlets are multi-threaded. A single servlet instance is typically created for each servlet class. The servlet container or web server will instantiate and initialize a servlet before passing requests to its `service` method. When a servlet container instantiates a servlet, the initialization methods in the servlet are executed.

A servlet can have two styles of initialization methods. An `init` method that is overridden from the `HttpServlet` class and an annotated method with any name. There are also two types of methods called before discarding the servlet.

Code 4-6 Example Initialization and Destruction Methods for a Servlet

```
public void init() {....}
@PostConstruct public void myInitMethod() {...}

public void destroy() {...}
@PreDestroy public void myDestroyMethod() {...}
```

Note – A single servlet classes can have several instance created at the same time if it is used by multiple `<servlet>` tags in the `web.xml` file.

Using the Request and Response APIs

The web container creates a request object and a response object for each new request that it handles. These objects are then passed to the service method of the servlet. In addition to providing information about the request, the request object allows the servlet to obtain user information and to pass data to other web components. The response object provides the servlet with mechanisms to generate a response or an error code to the browser.

The request and response APIs define over a hundred different methods, but there is only space to describe a few of them in this module. The methods that are described in this module are the ones that almost every servlet is likely to use. These methods perform the following actions:

- Retrieve form data
- Pass data from one component to another
- Retrieve security information
- Generate output

Request Object

An `HttpServlet` receives an object that implements the `HttpServletRequest` interface as its request object. `HttpServletRequest` is a subinterface of `ServletRequest`, so the servlet can call methods that are defined on either of these interfaces. Table 4-1 summarizes the most important methods that are exposed by the `ServletRequest` interface.

Table 4-1 Methods on the Generic Request Object

Method	**Purpose**
`getParameter`	Gets a request parameter, typically an element of form data that is sent by the browser
`setAttribute` and `getAttribute`	Sets and gets request attributes • Attributes are named data items that are inserted into the request. • Attributes are typically used to transfer data between web components.
`getRequestDispatcher`	Gets a request dispatcher, which can then be used to transfer control to another component

Table 4-2 summarizes the most important methods that are exposed by the `HttpServletRequest` interface.

Table 4-2 Methods on the HTTP-Specific Request Object

Method	Purpose
`getUserPrincipal`	Gets the identity of the user, if authenticated
`getCookies`	Gets cookies sent by the browser
`getSession`	Gets the client's session

Response Object

An `HttpServlet` receives an object that implements the `HttpServletResponse` interface as its response object. `HttpServletResponse` is a subinterface of `ServletResponse`, so the servlet can call methods that are defined on either of these interface. Table 4-3 summarizes the most important methods that are exposed by the `ServletResponse` interface.

Table 4-3 Methods on the Generic Response Object

Method	Purpose
`getOutputStream`	Gets an output stream for sending byte-based data to the client
`getWriter`	Gets an output writer for sending blocks of character-based data to the client
`setContentType`	Tells the client the MIME type of the data that it should expect to receive in the response body

Table 4-4 summarizes the most important methods that are exposed by the `HttpServletResponse` interface.

Table 4-4 Methods on the HTTP-Specific Response Object

Method	Purpose
`encodeURL`	Adds a session ID to an arbitrary URL The `encodeURL` method is required to support session management.
`addCookie`	Adds a specific cookie to the response

Table 4-4 Methods on the HTTP-Specific Response Object (Continued)

Method	Purpose
`sendError`	Sends an HTTP error code Use the `sendError` method when the servlet is unable to generate a response body.

Example of Handling Form Data and Producing Output

Code 4-7 shows a snippet that demonstrates how to read an element of form data, and how to form an HTML response in a servlet.

Code 4-7 Example of Reading Form Data and Producing Output

```
1   public void processRequest (HttpServletRequest request,
     HttpServletResponse response)
2         throws IOException {
3     response.setContentType ("text/plain");
4     PrintWriter out = response.getWriter();
5     String name = request.getParameter ("name");
6     if (name == null || name.length() == 0)
7       name = "anonymous";
8     out.println ("Hello, " + name);
9     out.close();
10    }
```

Although this is just a basic example, most servlets that produce output follow the same steps that are shown in Code 4-7 on page 4-14:

- Set the content type of the response that is delivered to the browser
- Get an output stream to write back to the browser
- Process form data
- Generate the response body exactly as the browser should receive it, which is in HTML, in this case

In addition, you should note the following things about Code 4-7 on page 4-14:

- The `getWriter` method throws an `IOException` if an output stream cannot be initialized.

  The servlet cannot continue in this situation, because it cannot produce any output, even an error message. Typically, the request processing method throws this exception out to the container, which generates an error message to the browser. `IOException` and `ServletException` are the only exceptions that a request handler is allowed to throw.

- The `getParameter` method returns `null` if the parameter is not present in the request at all, and returns a zero-length string if a parameter is present, but empty.

  If the user is presented with a form, but fails to fill in all of the fields, then the servlet should expect zero-length parameters and not null values.

- The `PrintWriter` buffers in blocks. Because of this, there is a risk that if the servlet does not produce a full block (typically eight kilobytes), the buffer will not be flushed when the service method completes.

  In this case, the browser receives no output at all. The servlet should ensure that the writer is flushed or closed when the service method is finished.

Forwarding Control and Passing Data

Software management is made easier when the application's logic is separated from the presentation of data to the user. This means that modern development practice favors the use of separate components for processing and presentation. Because no single component will carry out both processing and presentation, a servlet that is designed to do processing, rather than presentation, completes the following actions:

- Does its work and gathers data to be rendered
- Puts the data into the request
- Transfers control to the presentation component with the use of a `RequestDispatcher` object

The `RequestDispatcher` Interface

There are two ways to receive a `RequestDispatcher` implementation with a URI.

Code 4-8 The `ServletRequest.getRequestDispatcher("URI")` method which accepts relative paths or paths beginning with a `"/"`.

```
RequestDispatcher requestDispatcher =
request.getRequestDispatcher("relativeURI");
```


Code 4-9 The `ServletContext.getRequestDispatcher("URI")` method which must begin with a "/".

```
RequestDispatcher requestDispatcher =
  getServletContext().getRequestDispatcher
    ("/ServletName");
```

A servlet can be configured with a name but without a URI. This is done to prevent direct external access to a servlet. To use a `RequestDispatcher` to forward or include a servlet with no URI use the `getNamedDispatcher` method as shown:

```
RequestDispatcher requestDispatcher =
  getServletContext().getNamedDispatcher
    ("ServletName");
```

The `RequestDispatcher` Target and the Context Root

The argument to `getRequestDispatcher` is a URI, it is interpreted by the web container with reference to the current application's context root. That is, consider the following example, where the servlet that is getting the request dispatcher was invoked using a URI:

```
http://www.mybank.com/bank/Controller
```

In this case, the URI that is supplied to `getRequestDispatcher` is assumed to be within the application whose context is as follows:

```
http://www.mybank.com/bank
```

According to the servlet specification, a URI that is passed as an argument to `ServletContext.getRequestDispatcher` must begin with a slash (/), but this URI should not contain a context root or be a full URI. Developers often find this confusing, because the leading slash gives the impression that an absolute URI is being specified. It might help to consider that the leading slash is a short-cut for the context root. The presence of the slash is what prevents the need to give the context root itself.

In the bank sample application, the servlet obtains the JSP component to which it will transfer control using the following statement:

```
getRequestDispatcher ("/showCustomerDetails.jsp");
```

The URI begins with a slash, but this is simply a short-cut for the `/bank` context root. The `ServletRequest.getRequestDispatcher` method can take a relative URL as long as it is within the application context.

The `forward` and `include` Methods

The `RequestDispatcher` interface provides two methods to transfer control:

- The `RequestDispatcher.forward` method – The output of the component to which control is transferred becomes the output of the component that invoked it. The invoking component cannot produce any output of its own. Typically used by controllers.
- The `RequestDispatcher.include` method – The output of the component to which control is transferred is inserted into any output that is generated by the invoking component. Typically used by views.

Of the two transfer methods, `forward` is slightly faster, but cannot be used to merge the output of one component into the output of another component.

Note – The `forward` method cannot be used if the servlet has already begun to produce output. This means that after the servlet has called the `response.getWriter` method, it is too late to use the `forward` method, regardless of whether the servlet has used the writer itself. This is because the web container might already have sent some header information back to the browser.

Transfer of Data in the Request Object

The request object can carry data from one component to another component with the use of the following procedure:

1. In the invoking component, gather the required data.
2. Before the servlet transfers control, it adds the data as a named attribute to the request as follows:

```
CustomerData customerData = // get customer data
request.setAttribute ("customerData", customerData);
requestDispatcher.forward (request, response);
```

3. In the component that is being invoked, retrieve the named attribute, which contains the data:

```
CustomerData customerData = (CustomerData)
  request.getAttribute("customerData");
```

If the invoked object is a JSP component, the same effect can be achieved as follows, without embedding Java programming language statements:

```
<jsp:useBean id="customerData"
  class="bank.CustomerData" scope="request"/>
```

Note – `jsp:useBean` is covered later in Module 5.

Using the Session Management API

The Java EE platform uses a basic session management model that is based on the `HttpSession` interface. The principle features of the session API allow a servlet to perform the following operations:

- Determine whether a session has just been created
- Add a named item to the session
- Retrieve a named item from the session
- Close the session

The web container implements the Java EE session management functions and provides each client (browser) with its own private storage area on the server. This storage area is an object that implements the `HttpSession` interface. It behaves essentially the same as a `Hashtable` object into which the servlet can store named items of data. When the web component retrieves a `HttpSession` object, the container ensures that the web component obtains the correct object for the client whose request is being serviced.

To make it possible for the server to identify the client from one request to the next, one of the items stored in each session is a *session identifier*, called a session ID. Each new browser gets its own unique session ID when its session is first created. The session ID is sent to the browser, along with the response from the server, and the browser then presents the session ID on each subsequent request.

Figure 4-4 shows the web-tier session management model that is used by the Java EE platform.

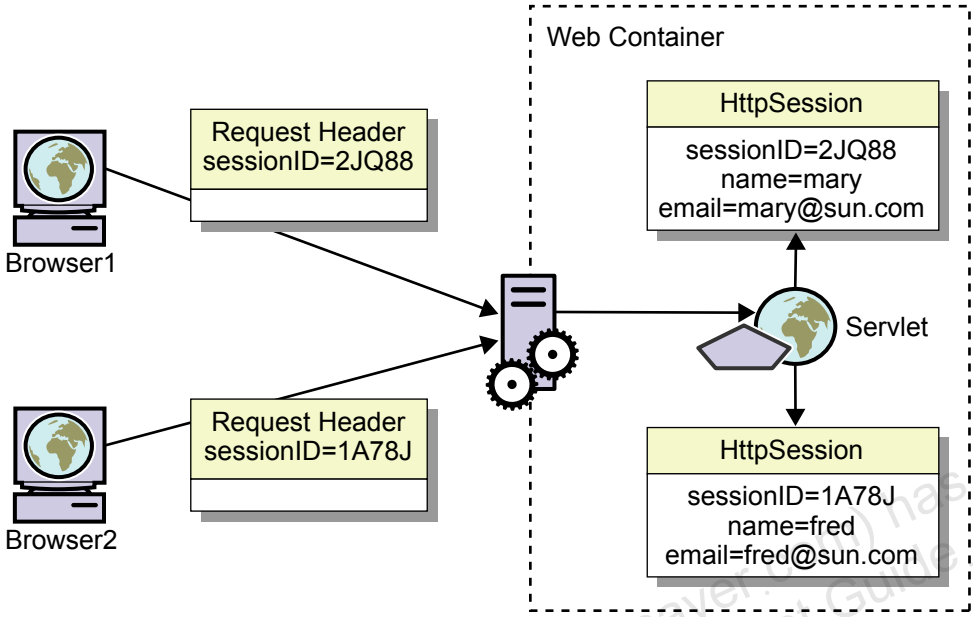


Figure 4-4 Java EE Platform Web-Tier Session Management Model

Session and Authentication

A Java EE web component can store whatever data is necessary in the session object. However, the web container also uses the session to store internal information about the session. For example, after a user has been authenticated by the web container, the user's ID and authentication status becomes part of the session. Consequently, the user has to log in only once in each session. After the user has authenticated, a servlet developer can call the `request.getUserPrincipal` method to get the user's login identification.

Session Binding

Session binding is the name of the technique that is used to support the binding of a session to a browser client. There are two main session-binding techniques:

- Session binding using cookies – The container reads and writes the cookies so there is no additional work for you. The use of cookies is the preferred session binding technique, but it is limited by the fact that not all browsers support cookies.
- Session binding using URL rewriting – You must ensure that a session ID is appended to every URI that the browser sees. For example, if the web component generates HTML like the following fragment, the '`showaccounts`' URI must be rewritten to expose the session ID:

  ```
  <A HREF='showaccounts'>Show accounts</a>
  ```

  The following code shows how the '`showaccounts`' URI would be rewritten:

  ```
  <A HREF='showaccounts;jsessionid=147JQ'>Show
  accounts</a>
  ```

  The web component API provides an `HttpResponse.encodeURL(String url)` method to carry out this conversion. However, you must ensure that it is called on every generated URL.

Note – Developers often mistakenly believe that `encodeURL` encodes form data to protect against interpretation by the browser, meaning that it converts spaces to + signs and punctuation to hexadecimal. This is not the case. This conversion is the function of the `java.net.URLEncoder` class.

Session Timeout

A well-designed web application gives the user an opportunity to close the session (to log out, for example). However, users do not always take this opportunity. In any case, if the web browser or workstation crashes, it might not even be possible to log out. Because the HTTP is stateless, the server has no way of knowing that a particular browser is no longer using a particular application.

To prevent its memory from becoming cluttered with unused and unusable sessions, the web container times out idle sessions after a period of inactivity. Most products use a default timeout of 30 minutes to a few hours, although this can be controlled both in the web application's deployment descriptor and in the code.

When a browser's session has timed out, a servlet or JSP component that retrieves a session for that browser obtains what appears to be a new session. A web application is generally not able to distinguish between a session that has timed out from a genuinely new session. You must ensure that this situation is handled, typically by reinitializing the session data when a new session is detected.

Session Lifetime

The `HttpSession.isNew` method returns `true` in either of the following situations:

- The session is genuinely a new session with a new browser.
- The current browser session timed out before this request.

In general, the web container cannot tell the difference between a post-timeout request and a new session. Therefore, the responsibility for discriminating between these two situations falls on you.

A session can be discarded before it times out as shown in Code 4-10.

Code 4-10 Invalidating a Session

```
1   if ("logout".equals(request.getParameter("action")) {
2           session.invalidate();
3   }
```

Code 4-11 demonstrates how to get the session object for the current client from the request, and how to initialize the session object at the start of the session:

Code 4-11 Retrieving a Session Object

```
1   // Get a session object for the current client,
2   // creating a new session if necessary
3   HttpSession session = request.getSession();
4
5   // If this is a new session, initialize it
6   if (session.isNew()) {
7     // Initialize the session attributes
8     // to their start-of-session values
9     session.setAttribute ("account", new Account());
10    // ... other initialization
11  }
12
13  // Get this client's 'account' object
14  Account account = (Account) session.getAttribute("account");
```

Summary

Servlets are web components that are expressed in the Java programming language. A servlet has access to an API, which allows access to the HTTP request data and the programmatic generation of an HTTP response. Modern practice favors the use of servlets for request processing and presentation management, while delegating the generation of content to JSP components. Although traditionally servlets have been seen as tools for generating dynamic web content, other applications have emerged recently. For example, a popular application is to use servlets to receive and produce XML data to support business-to-business operations.

Many additional servlet topics covered are covered in SL-314-EE6, Web Component Development with Servlet and JSP Technologies. Covered in detail are:

- Advanced form processing
- Advanced session management
- The cookie API
- Filters in web applications
- Lifecycle listeners

The Servlet 3.0 specification introduced new features that are beyond the scope of this course but are covered in SL-314-EE6 such as:

- Web fragments
- Asynchronous processing
- HttpOnly cookies

Module 5

Developing With JavaServer Pages™ Technology

Objectives

Upon completion of this module, you should be able to:

- Evaluate the role of JSP technology as a presentation mechanism
- Author JSP pages
- Process data received from servlets in a JSP page
- Describe the use of tag libraries

Additional Resources

Additional resources – The following references provide additional information on the topics described in this module:

- Eric Jendrock, Debbie Carson, Ian Evans, Devika Gollapudi, Kim Haase, Chinmayee Srivathsa. “The Java EE 6 Tutorial,” [`http://java.sun.com/javaee/6/docs/tutorial/doc/`], accessed 1 August 2009.
- “JavaServer Pages Technology.” [`http://java.sun.com/products/jsp/`], accessed 1 August 2009.

JSP Technology as a Presentation Mechanism

JSP pages are text-based documents that describe how to process a request and create a response. Using JSP technology, a page designer creates a document to generate a complex response that contains dynamic content.

In addition to static markup (HTML or XML), a JSP page contains JSP elements that are defined by the JSP specification. JSP elements include *actions*, *directives*, and *implicit objects* that enable external object access and add *canned* programming capabilities when processing a request.

Because JSP pages are text-based documents, they can be edited with any standard text editor. However, special editors add capabilities for syntax checking, compilation, and testing output generation. Source files for JSP pages typically end with the `.jsp` extension.

JSP technology has Write Once, Run Anywhere™ properties. A JSP page is written using a technology that is not proprietary, it can run on any web server, and it can be accessed from any web browser. You do not concern yourself with platform-specific issues when you create or deploy JSP technology. This feature also allows for a greater choice when you select appropriate JSP technology tools and components. Furthermore, the components accessed or used within a JSP page, such as components that are based on the JavaBeans™ component architecture (or beans), EJB components, and tag libraries, are all designed to be reusable components.

JSP technology uses beans to interact with server-side objects, and uses tag libraries to develop and extend the canned capabilities that are provided by *actions*. JSP technology allows for a high degree of separation between the static and dynamic content in a JSP page.

Where necessary, Java technology provides a powerful, platform-independent scripting language for JSP pages, which allows access to the entire suite of Java APIs.

Finally, JSP technology is an integral part of the Java EE platform, and so provides front-end access to EJB components.

Presentation Using JSP Pages Compared to Servlets

JSP pages are web components that are based on the servlet model. JSP pages implement the same run-time behavior as servlets. In particular, web browsers interact with JSP pages by using the same request and response model over a protocol, such as HTTP. In practice, JSP pages are translated into servlets.

JSP technology provides capabilities similar to other dynamic content response generation technologies, such as Active Server Pages (ASP), server-side JavaScript™, and Common Gateway Interface (CGI).

When you develop a JSP page you use a markup language that is similar in syntax to HTML. JSP technology has several distinct advantages over other dynamic content technologies, as well as advantages over servlets:

- Because a JSP page is compiled and runs as a servlet, it has similar runtime benefits over other scripting tools, such as CGI.
- Unlike servlets, JSP technology includes the ability to use a markup language outside of a series of `println` statements.
- JSP technology provides automatic recompilation of modified pages.
- JSP page authors are not required to be software developers. Authors can concentrate on how a page appears to a user, and use familiar tags to generate dynamic data. No coding is required.
- JSP pages are often quicker and easier to debug for presentation errors than servlets. JSP pages can be redeployed quickly, and their format makes it easier to see errors than during the process of debugging servlets for the same type of errors. Compile-time and runtime errors in JSP pages can be more difficult to debug as most error messages give line numbers in the container generated servlet.

Worker Beans, JSTL, and Custom Tags

A web-based user interface that uses JSP components should not embed Java programming language statements because this practice makes the presentation elements more difficult to manage. You can separate programmatic functionality from presentation in JSP components in two ways:

- Incorporate classes using the `<jsp:useBean>` tag and the JavaServer Pages Standard Tag Library (JSTL)

  This technique is useful for carrying data into the JSP component. It can also be used to embed application logic. For example, a class that is incorporated using the `<jsp:useBean>` tag can validate form data and control program flow. However when bean classes are incorporated for application logic, they tend to become tightly coupled to the JSP component and hard to maintain.

  JSTL allows programmatic behavior without the use of scriptlet code blocks.

- Make use of custom tag libraries

  The development overheads of custom tags mean that they are of most value when they are general-purpose and reusable. Custom tag libraries are less useful for application-specific logic, such as form processing. It is time-consuming to develop custom tags that are tightly coupled to a JSP component and unlikely to be used in a different project.

- Both `useBean` classes and custom tags have a part to play in the development of web applications. However, experience has shown that they are not sufficient on their own to implement the full functionality of the web tier without detriment to maintainability.

JSP Page Deployment Mechanism

As mentioned, JSP pages are translated into servlets. This translation must occur at some point before the JSP page is invoked. In principle, the translation could be done in various ways. For example, it could be the developer's responsibility to use a tool that is provided by the vendor to do the translation before deployment.

Such tools do exist, and are widely used. However, the Java EE specification states that a web container that supports JSP technology must be able to *translate a JSP page on demand*.

JSP technology is designed to simplify presentation design, and must be manageable by content authors, as well as by developers. The Java EE specification takes the approach that a JSP page can be deployed in the same manner as an HTML page, by copying the file onto the server in the appropriate place.

Translate-on-Request Process of JSP Pages

As mentioned, a JSP container must be able to transform a JSP page into a servlet, and then compile and load the servlet before executing the page. The JSP container does not need to go through this process on every request. The JSP container recognizes when a JSP page has been updated since the last request, and recompiles pages as necessary. Automatic page recompilation provides a robust mechanism for maintaining pages that require frequent modification. Figure 5-1 illustrates the page translation procedure for JSP pages.

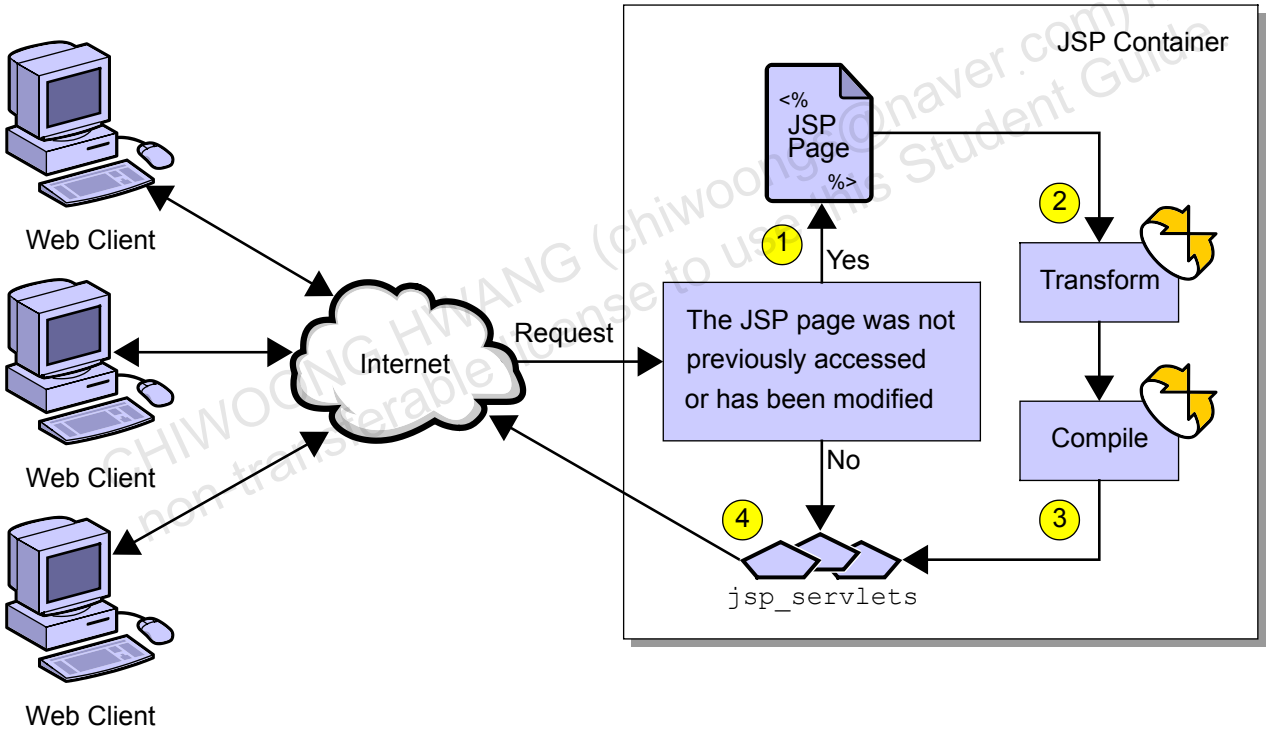


Figure 5-1 JSP Page Translation Procedure

Java Code Embedded in JSP Pages

The JSP specification also defines the requirements for a scripting language that is used within a JSP page. Instead of identifying a specific scripting language that must be used, the specification lists some basic capabilities that a scripting language for JSP technology must provide. If no scripting language is specified, the Java programming language is used by default.

As a page designer you can present dynamic data within a JSP page with minimal effort using only standard JSP elements. You can also incorporate scripting code within a JSP page to perform advanced processing logic and flow control as necessary. There are multiple problems inherent in incorporating scripting code for processing logic and flow control within a JSP page:

- It requires a JSP page author to know how to code well in the scripting language.
- It might require a JSP author to have more business domain knowledge.
- It is more difficult to see presentation information when you view the JSP page.
- It is more difficult to debug because of the added complexity and decreased clarity.

Ideally, a JSP page should be concerned with presentation logic only. A servlet is a better alternative for processing logic and flow control.

Authoring JSP Pages

A JSP page contains standard markup tags, such as HTML or XML, associated text data, and a variety of elements that are defined by the JSP specification. Figure 5-2 shows the various components of a JSP page.

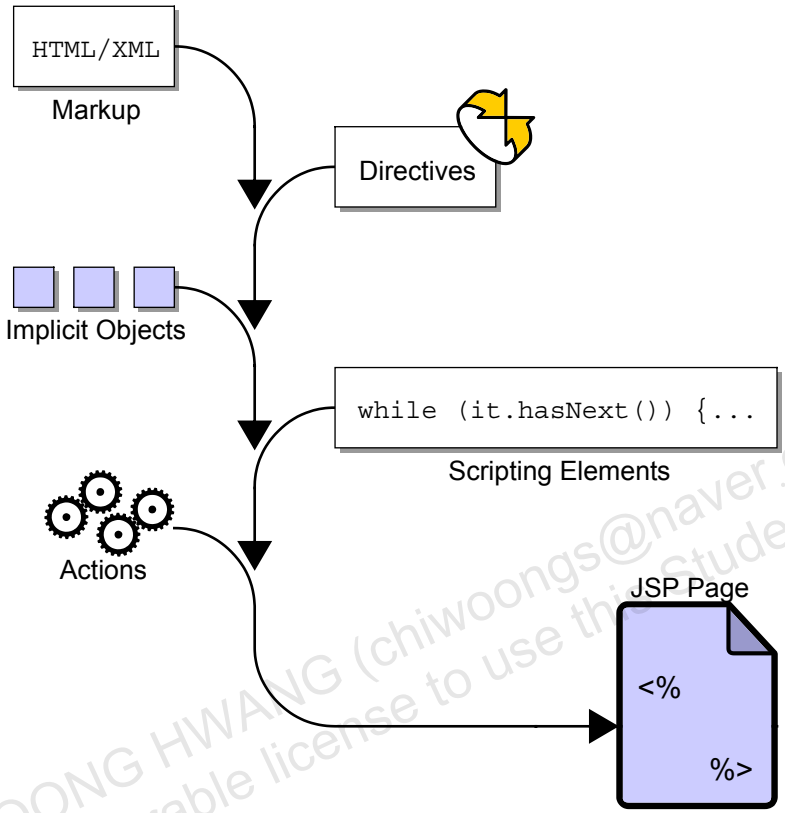


Figure 5-2 JSP Page Components

This course covers a subset of these components. Detailed coverage of JSP pages can be found in SL-314-EE6, Web Component Development with Servlet and JSP Technologies.

Syntactic Forms of JSP Tags

Syntactic forms of JSP tags can be represented in two different ways:

- The first style has been part of the JSP specification from the beginning and is similar to other tag-based dynamic presentation technologies, such as ASP.
- The second style is similar to XML, with beginning and ending tags.

Table 5-1 gives examples of the two styles of tags.

Table 5-1 Syntactic Forms of JSP Tags

Old Syntax	XML Syntax
`<%! ... %>`	`<jsp:declaration> ... </jsp:declaration>`
`<%= ... %>`	`<jsp:expression> ... </jsp:expression>`
`<% ... %>`	`<jsp:scriptlet> ... </jsp:scriptlet>`
`<%@ ... %>`	`<jsp:directive.`*`type`*` ... />`

The JSP 2.1 specification specifically supports the XML syntax for tags. Some, but not all prior implementations also support the XML syntax for tags.

JSP Technology Directives

A JSP technology *directive* contains information to help a JSP container configure and run a JSP page. The information that is contained within a directive is associated with the compiled servlet that is created from the JSP page, and all requests to the JSP page share this information. Directives do not produce any output into the current `out` stream. The generic syntax for a JSP page directive is:

```
<%@ directive attribute="value" ... %>
```

Note – The `out` stream is one of several *implicit objects* that all JSP pages can access. The `out` stream represents a stream that displays information to the client. Implicit objects and their uses are presented in depth in SL-314-EE6, Web Component Development with Servlet and JSP Technologies.

The *directive* is the directive type and the *attribute* and *value* pair is one or more of the applicable combinations for the defined directive type. Three directive types are defined in the current JSP specification:

- The `page` directive
- The `include` directive
- The `taglib` directive

Figure 5-3 illustrates the three directive types.

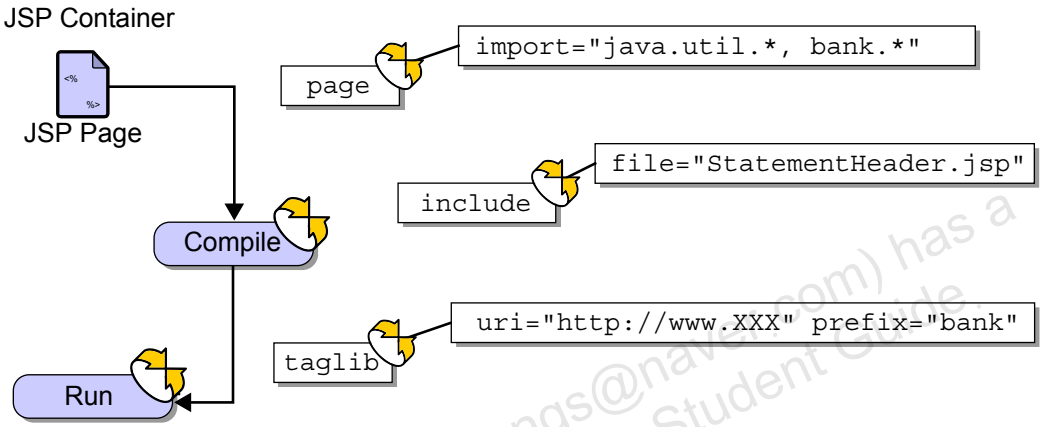


Figure 5-3 JSP Directives

Although directives are generally position-independent, directives that affect how a JSP container handles a page are often specified on the first line of a JSP page for clarity.

The `page` Directive

The `page` directive defines page-dependent attributes. An attribute and value pair cannot be redefined within a *translation unit*, with the exception of the `include` page directive. Redefining a `page` directive results in a fatal translation error.

Note – A translation unit is defined as a JSP page source file and any files included through the `include` directive.

The following `page` directive attributes are defined by the current JSP specification:

- The `language` attribute indicates the scripting language used.
- The `session` attribute specifies whether the page participates in a session.
- The `buffer` attribute specifies the buffering model for the `out` stream.
- The `errorPage` attribute defines a URL used to forward `Throwable` objects that are not handled by the page for error processing.
- The `import` attribute describes an import list of types that are available for use by the scripting environment.
- The `isErrorPage` attribute indicates that the page is the target of another page's `errorPage` directive when set to `true`.
- The `isThreadSafe` attribute sets the level of thread safety implemented in the page.
- The `info` attribute contains a string included into the translated page that can contain some information about the page implementation.
- The `autoFlush` attribute determines what happens when the output buffer is filled.
- The `isELIgnored` attribute determines if EL expressions are ignored or translated.

Refer to the JSP 2.1 specification and syntax sheet for a complete listing of page directives, attribute usage, syntax, and default values. The following are examples of the page directive using both styles of syntax:

```
<%@ page import="java.util.*, java.lang.*" %>
<%@ page buffer="5kb" autoFlush="false" %>
<jsp:directive.page errorPage="error.jsp" />
```

Figure 5-4 illustrates many of the page directives.

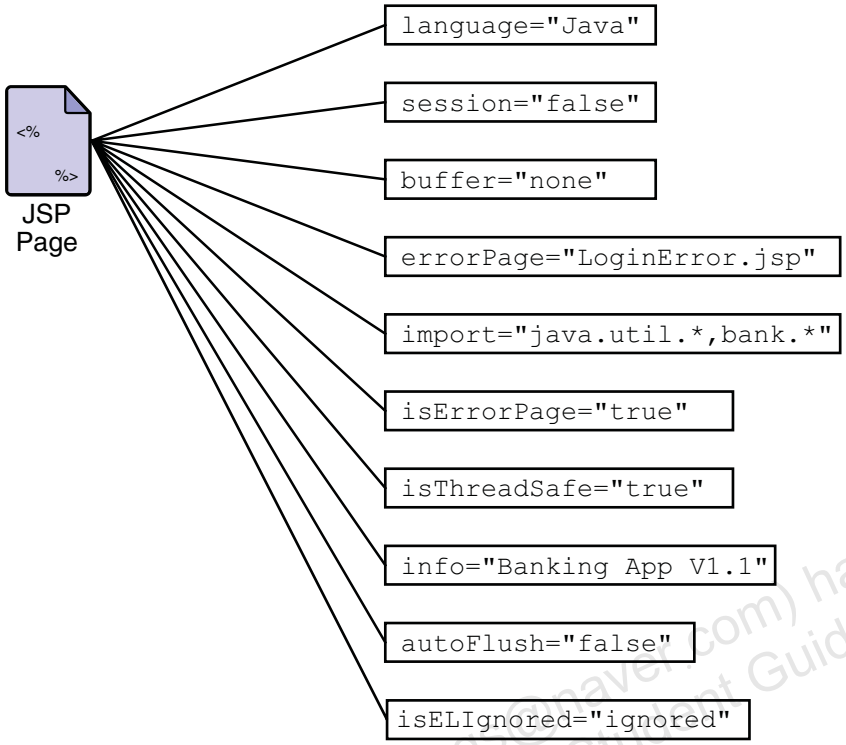


Figure 5-4 JSP `page` Directives

The `include` Directive

The `include` directive inserts the text of the specified resource into the `.jsp` file at page translation time. Resources that are specified using the `include` directive are treated as static objects. The text of the included resource is inserted into the page at the position of the `include` directive at page translation time. Included resources can be other HTML files or other JSP pages that contain text, or code, or both. The attribute and value pair for the include directive points to the included resource, as shown in the following examples:

```
<%@ include file="relativeURL" %>
```

Or

```
<jsp:directive.include file="relativeURL" />
```

Changes to included resources might not be recognized by the JSP container and, therefore, might not force automatic recompilation of the including page.

Figure 5-5 illustrates the use of an `include` directive to include static HTML heading information.

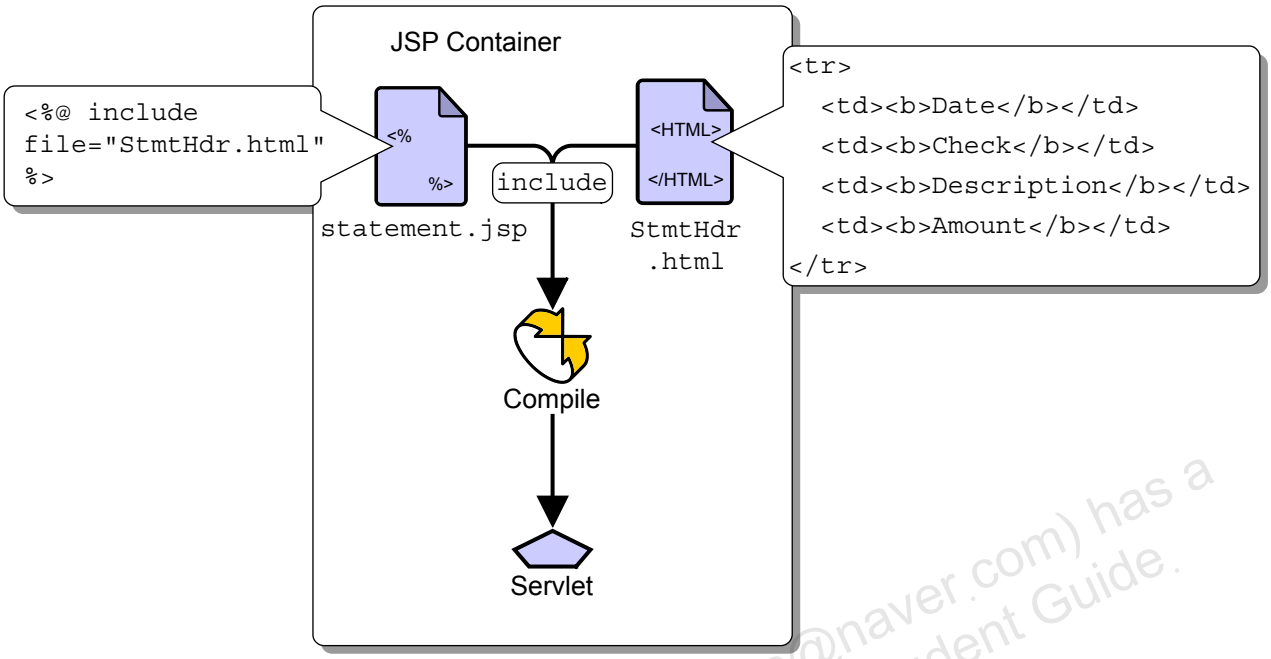


Figure 5-5 The `include` Directive

Declarations, Expressions, and Scriptlets

Special scripting elements allow a page designer to embed code in a page to provide advanced programming capabilities. Scripting elements include:

- *Declarations* – Used to declare variables and methods that can be accessed from within a page
- *Expressions* – Evaluated by the JSP container at runtime
- *Scriptlets* – Used to embed snippets of scripting code

Declaration Scripting Elements

The code inside a declaration tag must conform to the syntax of the specified scripting language and form complete declarative statements in that language. In practice, this means that the code must be valid Java code. The current JSP specification only defines the `Java` value for the scripting language.

The declaration is associated with the body of the compiled servlet and, therefore, variables declared this way have *instance* scope. On the whole, declarations of methods, constants, and variables, which are only assigned values during initialization, should not be problematic. By their nature, declarations do not produce output into the current `out` stream.

The syntax for a declaration is as follows:

```
<%! declaration(s) %>
```

Or

```
<jsp:declaration>
declaration(s)
</jsp:declaration>
```

Code 5-1 illustrates the declarative syntax.

Code 5-1 Declarative Syntax

```
<%! final String SHOWDETAILS_URL = "/showdetails.jsp";
boolean hasAccounts(Customer c) { return !c.getAccounts().isEmpty(); } %>
<jsp:declaration>
  // This instance variable is assigned at initialization time
  protected BankMgr bankMgr = null;
</jsp:declaration>
```

Expression Scripting Elements

An *expression* is any legal expression, as defined by the specified scripting language. The JSP container evaluates expressions at runtime. The JSP container automatically converts the result of an expression to a `String` object and inserts the string into the page. Expressions are useful for retrieving values of page variables, methods, or bean fields for inclusion in the generated response.

The syntax for an expression element is as follows:

```
<%=expression %>
```

Or

```
<jsp:expression>
expression
</jsp:expression>
```

For example, the following line inserts the value returned by the `getBalance` method into an HTML table cell. The table cell is indicated by the HTML element tag `<td>`:

```
<td> <%=acct.getBalance()%> </td>
```

Or

```
<td>
<jsp:expression>
acct.getBalance()
</jsp:expression>
</td>
```

Scriptlet Scripting Elements

Scriptlets are raw blocks of program code. The translator for JSP pages (JSP translator) inserts scriptlets into the generated servlet without modification. When scriptlets use Java technology as the scriptlet language, the scriptlet has full access to the complete Java API set. If necessary, you can use the `import` attribute of the `page` directive to include any additional libraries that are required by your code.

The JSP specification requires support for scripting elements based on the Java programming language. In principle, other scripting languages can be implemented as long as they support manipulation of objects, invocation of methods on objects, and catching exceptions. In practice, the Java language is invariably used.

A scriptlet can contain any code segment that is valid for the specified scripting language. The syntax for inserting a scriptlet into a JSP page is as follows:

```
<% code_segment %>
```

Or

```
<jsp:scriptlet>
code_segment
</jsp:scriptlet>
```

Code 5-2 shows the *code_segment* section of the script.

Code 5-2 The *code_segment* Section

```
1   <%
2      if (isAllowedTransaction() == false) {
url=ScreenMgr.BANK_ERRORPAGE; }
3      else {
4         Vector checkList = account.getCheckByAmount(amt);
5         Iterator it = checkList.iterator();
6         double totalCheckAmount = 0.00;
7         while (it.hasNext()) {
8            Check chk = (Check) it.next();
9            totalCheckAmount += chk.amount();
10        } // end while
11     } // end if/else
12  %>
```

You can intermingle code segments with markup components and other valid JSP elements, as long as the code fragments are properly delimited using <% and %>. All code fragments, when combined into a complete segment in the order in which they appear in the page, must create a valid, compilable code segment according to the syntax rules of the specified scripting language, as shown in Code 5-3.

Code 5-3 Combining Code Fragments Into a Code Segment

```
1   <%
2      Check chk;
3      while (it.hasNext()) {
4         chk = (Check)it.next();
5         // end of first code fragment
6   %>
7         <%-- output check amount using HTML --%>
8         <br> Check Amount: <%=chk.getAmount()%> </br>
9   <%
10     } // closing bracket for while loop
11        // end of second code fragment
12  %>
```

After translation, when all code fragments are combined, the code illustrated in Code 5-3 results in the legal servlet code that is shown in Code 5-4.

Code 5-4 Legal Servlet Code After Translation

```
1   Check chk;
2   while (it.hasNext()) {
3      chk = (Check)it.next();
4      // end of first code fragment
5      out.write("\t\t<br> Check Amount: ");
6      out.print(chk.getAmount());
7      out.write(" </br>\r\n");
8      out.write("");
9   } // closing bracket for while loop
10  // end of second code fragment
```

Use scriptlets sparingly. Besides being difficult to read, the use of scriptlets does not support the separation of Web-tier roles and code reusability. Custom tag libraries are an excellent alternative to scriptlets.

Thread-Safety Implications

Declarations, and anything else that is defined as instance scope, occur at the instance level of the generated servlet. This means that all requests to the JSP page share these variables and methods, and you must be careful not to cause thread-safety problems with this technique. All of the cautions that apply to servlets and thread-safety apply to JSP pages, and specifically to JSP page declarations.

Processing Data From Servlets

Server-side *actions* provide a *canned* range of operations to the JSP page developer. Actions use an XML-style tag format to allow a JSP page developer to use, modify, or create objects, without requiring a detailed understanding of the Java programming language. These objects might include data that is passed from a servlet. Based on the specific action type, an action might affect the current output stream, that is, produce content.

The JSP specification defines a standard set of action types that all conforming JSP containers must implement. The following sections list some of the standard actions that are available to complete the following types of actions:

- Create or use beans
- Set and get bean properties
- Include static and dynamic resources in the current page's context

You can define additional action types as necessary using custom tag libraries that are identified with the `taglib` directive.

The syntax for an action includes an XML-style action tag and an associated list of attribute and value pairs. For example, the action syntax to create or use a bean within a JSP page is:

```
<jsp:useBean id="name" scope="scope" typeSpec />
```

A brief summary of the standard action types defined in the JSP 2.1 specification follows.

Note – For a more complete explanation of the range of operations and syntax of each action, refer to the JSP specification or to SL-314-EE6, Web Component Development with Servlet and JSP Technologies.

The `jsp:useBean` Action

The `jsp:useBean` action creates or locates a bean instance, and associates it with a scope and action ID so that the instance is accessible by scripting elements and custom tags. In this case, the *bean* is a standard JavaBeans component, *not* an EJB component. Before it creates an object instance, the `useBean` action attempts to locate an existing object with the specified ID and scope attributes. If a bean instance matching the criteria is found, the existing bean's object reference is used instead of creating a new bean instance. The following is the syntax to create or use a bean within a JSP page:

```
<jsp:useBean id="name" scope="scope" typeSpec />
```

In this syntax example, the *name* variable has a specific value and the *scope* variable is one of the four valid JSP scopes, including `page`, `request`, `session`, or `application`. The *typeSpec* attribute and value pair determines the specific steps taken to locate or create a bean instance. The *typeSpec* parameter can be one of a number of legal attribute and value pairs for this action that specify either the class or type of the object, a combination of class and type, or combination of type and bean name. Figure 5-6 illustrates the `jsp:useBean` action.

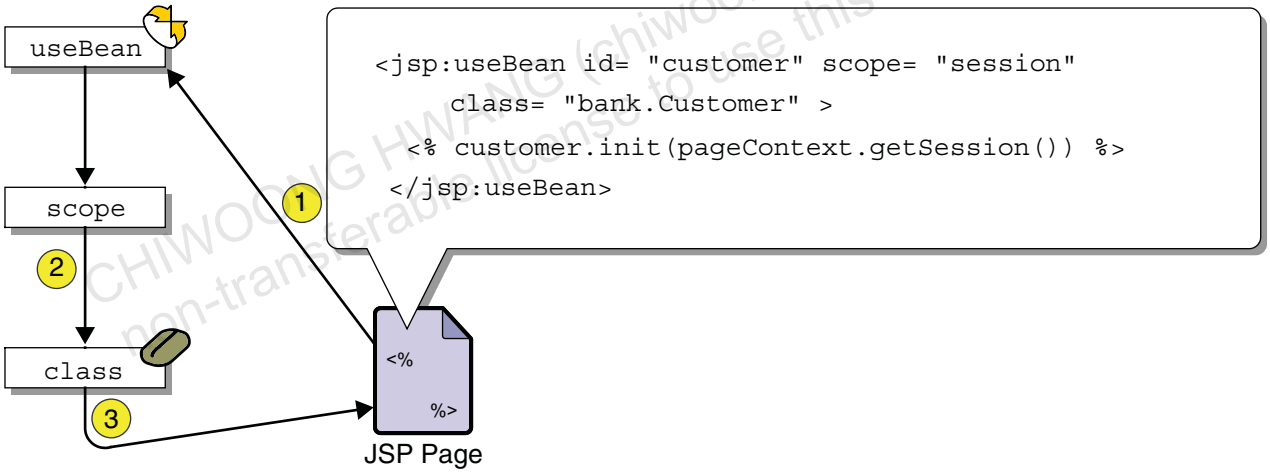


Figure 5-6 The `jsp:useBean` Action

An alternative form of the `jsp:useBean` action allows you to specify a body that contains initialization code to execute if the bean does not already exist and must be created.

```
<jsp:useBean id="name" scope="scope" typeSpec >
  <% ...initialization code... %>
</jsp:useBean>
```

The `id` attribute in the `jsp:useBean` action identifies the object to the JSP container and page. The `id` attribute must be unique within its translation unit. In the Java programming language, `id` becomes the name of a method-scope variable in the generated servlet. You can use this name to access the object within a page from the scripting language. In addition, the JSP page can access object instances that are created by `jsp:useBean` through the `pageContext` object. Figure 5-7 illustrates the `id` attribute.

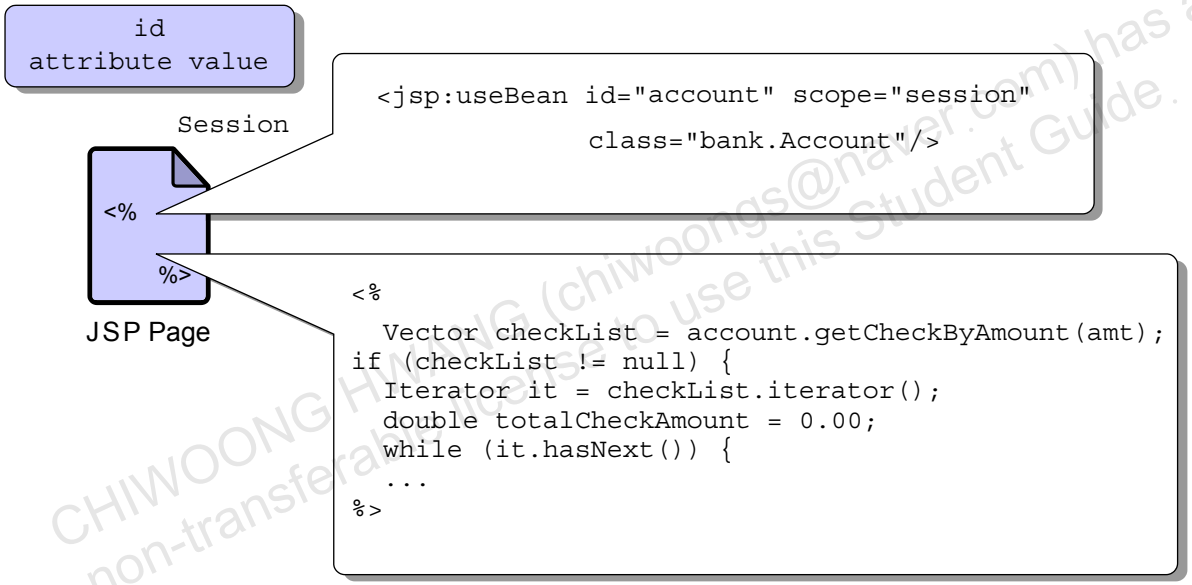


Figure 5-7 The `id` Attribute

For example, after you establish a bean reference by using the following code, you can retrieve an account's balance:

```
<jsp:useBean id="account" class="bank.Account"/>
```

You can use the following expression to retrieve an account's balance:

```
<%=account.getBalance() %>
```

Scope Rules for the `jsp:useBean` Action

The `scope` attribute determines the namespace, object reference life cycle, and accessibility of an object. Objects are always created in response to a request within an instance of a JSP page. When you create an object, you can request one of four distinct scopes, depending on the nature of the object and the desired visibility or life cycle. Figure 5-8 illustrates the various scope values for `jsp:useBean`.

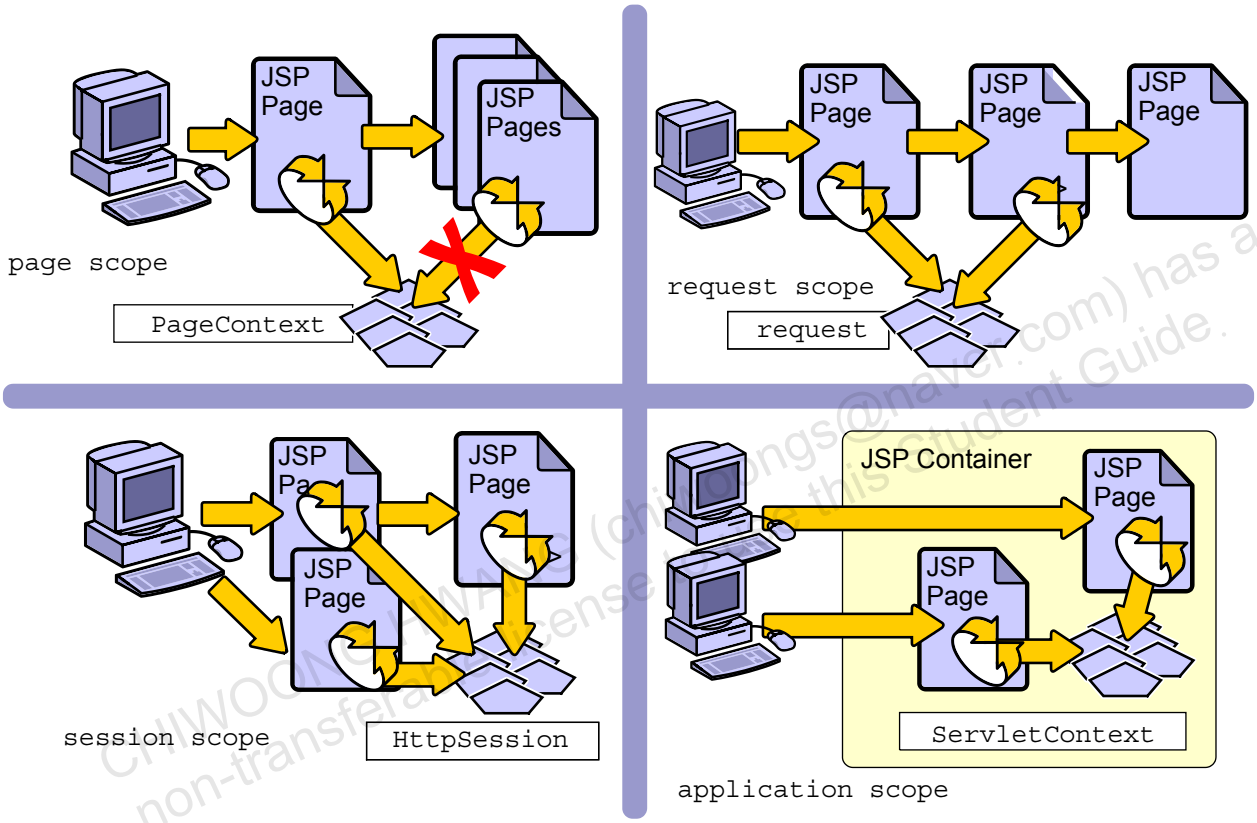


Figure 5-8 The `jsp:useBean` Scopes

The `page scope` attribute defines an object that is accessible only from within the page where it is created. All object references to objects with the `page scope` attribute are stored in the `PageContext` object for the current page and are released when the JSP page is done processing or the request is forwarded.

The `request scope` defines an object that is accessible from all pages that are processed during the specific client request in which they were created. A `request scope` object reference is stored with the request object and is released when the request is through processing, regardless of how many times the request is forwarded or how many JSP pages are created.

You can create `session scope` objects only on pages that are *session-aware*. Session-aware means that a session object is associated with this page. A `session scope` object is accessible from all pages that are associated with the same session object. In practice, this means that objects with the `session scope` are accessible from all requests from the same client in the same session. Object references with `session scope` are stored with the session object (`HttpSession`) and are released when the associated session ends or is invalidated.

The `application scope` object references are stored with the application object that is associated with a page activation and are accessible from all pages processing requests that are in the same application as the one in which they were created. Object references with `application scope` are released when the container reclaims the `ServletContext` object.

Request-Scope Beans and Collecting Data From Servlets

There are many uses for the `jsp:useBean` action, and one of the most useful applications is when you need to share data from servlets. Servlets often perform front-end processing in an application and subsequent dispatching to JSP pages to display the dynamic data. You set an attribute on the request object in the servlet, and you use the `jsp:useBean` action, with the `request scope` attribute, to collect the data from within the JSP page. For example, in the servlet code snippet that is shown in Code 5-5, you create a new `Customer` object and save it in a request attribute named `customer`.

Code 5-5 Creating a New Object and Saving it in a Request Attribute

```
1   public void doPost (HttpServletRequest request, HttpServletResponse
response) {
2      ...
3      try {
4         Customer cust = new Customer(firstName, lastName);
5         request.setAttribute("customer", cust);
6
7         // use a request dispatcher to forward to a JSP page
8         RequestDispatcher disp =
9         getServletContext().getRequestDispatcher("/jsp/example.jsp");
10        disp.forward(request, response);
11     } catch (Exception ex) {
12                  . . .
13     } // end catch
14   }
```

The JSP page `example.jsp` that is illustrated in the Code 5-6 snippet can then access and process the `customer` object after first using the `jsp:useBean` action to extract it from the default `request` scope.

Code 5-6 Using the `jsp:useBean` Action to Extract the Object

```
1   <jsp:useBean id="customer" class="bank.Customer" scope="request"/>
2   ...
3   <jsp:getProperty name="customer" property="firstName" />
4   <jsp:getProperty name="customer" property="lastName" />
5   ...
```

The `jsp:getProperty` action gets the value of a property for the named bean instance, converts it to a `String`, and places it in the implicit `out` object. The syntax for the `getProperty` action is:

```
<jsp:getProperty name="bName" property="pName" />
```

In this syntax example, the *bName* variable is the name of a previously created bean instance and *pName* variable is the name of the property to get. Object instances that are retrieved with the use of this action are converted to a `String` value using the object's `toString` method.

Figure 5-9 illustrates the use of the `jsp:getProperty` action.

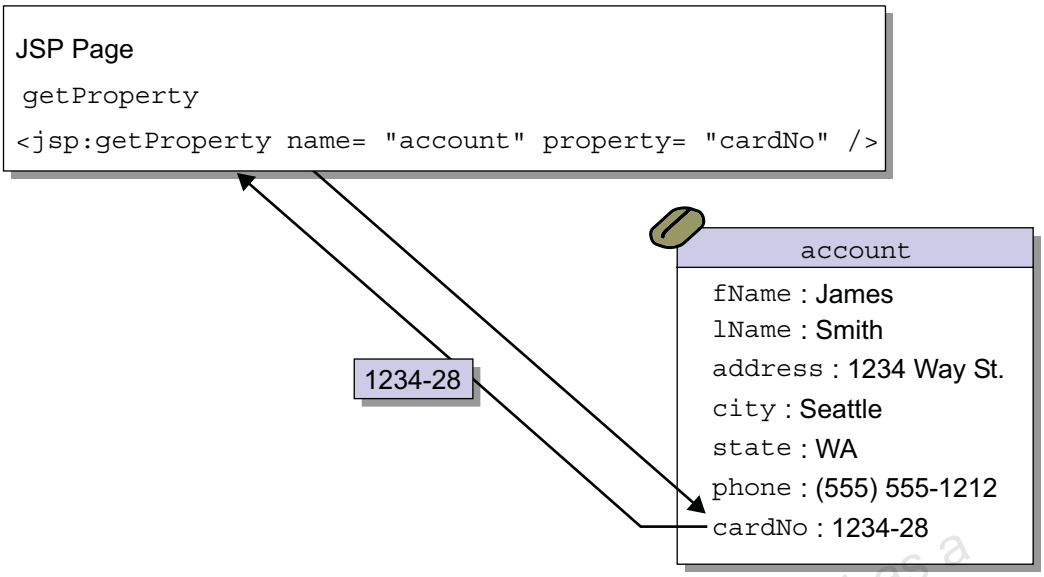


Figure 5-9 The `jsp:getProperty` Action

Custom Tag Libraries

Because it is preferable to avoid scriptlet code in JSP pages, another mechanism must exist for customizing JSP pages to suit your business needs. Custom tag libraries fill this need. Custom tag libraries supply standard or customized server-side actions that use an XML-style tag format to provide canned capabilities to use, modify, or create objects.

The JavaServer Pages Standard Tag Library (JSTL) is a standard library of tags that is available from `http://java.sun.com/products/jsp/jstl/index.jsp`. JSTL encapsulates, as simple tags, core functionality that is common to many JSP applications. The use of JSTL is not covered in this course.

The `taglib` Directive

The `taglib` directive extends the set of tags that a JSP container can interpret by associating a tag prefix with a tag library. A tag library is a set of classes and a *tag library descriptor*. A tag library implements the range of operations defined in the tags.

The format for the `taglib` directive is as follows:

```
<%@ taglib uri="URIToTagLibrary"  prefix="tagPrefix" %>
```

The following code examples and associated text show the use of a tag library that implements a general-purpose iterator. JSP page developers use the iterator tag to iterate over anything that implements the `java.util.Iterator` interface.

The following `taglib` directive tells the JSP translator that any tag that begins with `iterator:` has its range of operations defined in a custom tag library, which can be located using the identifier `iterator_tags`.

```
<%@ taglib uri="iterator_tags" prefix="iterator" %>
```

Code 5-7 illustrates how the iterator tag library might be used in a JSP page. The `iterate` functionality of the `iterator` tag comes from classes that are provided by the tag library.

Code 5-7 Custom Tag Library Usage

```
1   <iterator:iterate>
2      <%-- perform repetitive task --%>
3      ...
4   </iterator:iterate>
```

The `tag-library` Descriptor and Java Classes

The URI in the taglib directive maps a prefix to a TLD. The TLD is an XML file, which is usually packaged in the web application or library, along with the classes that implement its range of operations.

When the JSP translator encounters the following line in the JSP page, it knows which class provides the range of operations to support the `iterator:iterate` tag:

```
<iterator:iterate id="accounts">
```

This information comes from the TLD, which has a section like the one shown in Code 5-8 on page 5-28.

Note – For the Glassfish Application Server the JSTL TLD files can be found inside of a JAR file that is located underneath the App Server installation directory. These TLD files are detected and used automatically. The TLD files are located in the META-INF directory of `glassfish/modules/jstl-impl.jar`.

Code 5-8 Information From the TLD

```
1   <tag>
2      <name>iterate</name>
3      <tagclass>
4         com.acme.tags.IteratorTag
5      </tagclass>
6      <attribute>
7         <name>id</name>
8         <required>true</required>
9      </attribute>
10  </tag>
```

This `<tag>` element says that the tag `iterate` is supported by a class called `IteratorTag`. It also says that the `iterate` tag recognizes one attribute, `id`, which the JSP page must supply. In Code 5-8, `id` is the name of an instance that is already created within the JSP page, and perhaps passed in from a servlet in the request.

The TLD information allows the JSP translator to generate code that makes method calls on the class `IteratorTag`. These method calls are made repeatedly, until the `IteratorTag` signals that it has finished. This allows the implementation of a looping tag, which is exactly the range of operations that this example needs.

The Expression Language (EL)

The Expression Language (EL) is an easy to use language that can be embedded in JSP pages instead of scriptlets for the evaluation and retrieval of variables. The syntax of EL is similar to ECMAScript and therefore should be easy to leverage for web developers without a large Java background. EL can be combined with JSP tab libraries to completely remove the need for scriptlets in JSP pages.

Code 5-9 Expression Language examples

```
${ 3 + 2 }

${ param.address }

${ requestScope.customer.name }

${ not empty sessionScope.message }
```

Note – For a more complete explanation of the Expression Language refer to the JSP specification or to SL-314-EE6, Web Component Development with Servlet and JSP Technologies.

The JSTL Core Tag Library

Java EE 6 provides several pre-written custom tags, known as the JavaServer Pages Standard Tag Library or JSTL. These libraries are grouped by functionality. The most commonly used library is the *core* library.

The following tag libraries are part of the JSTL:

- The core library provides alternatives to scriptlet structures, such as if statements and for loops. The core library uses the `c` prefix within the JSP page and the identifying URI of the library is: `http://java.sun.com/jsp/jstl/core`.
- The XML processing library provides functionality similar to the core library for use with XPath expressions and XML transformation ability. The XML library uses the `x` prefix within the JSP page and the identifying URI of the library is: `http://java.sun.com/jsp/jstl/xml`.
- The I18n formatting library provides internationalization capabilities. The I18n library uses the `fmt` prefix within the JSP page and the identifying URI of the library is: `http://java.sun.com/jsp/jstl/fmt`.

- The relational database access or SQL library provides access to relational databases. The SQL library uses the `sql` prefix within the JSP page and the identifying URI of the library is: `http://java.sun.com/jsp/jstl/sql`.
- The functions library provides a standard set of functions. The functions library uses the `fn` prefix within the JSP page and the identifying URI of the library is: `http://java.sun.com/jsp/jstl/functions`.

Code 5-10 JSTL and EL Examples

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core"
prefix="c"%>

<c:forEach var="item" items="${requestScope.list}">
   <tr><td>${item.var1}</td><td>${item.var2}</td></tr>
</c:forEach>

<c:if test="${x < 3}" >

</c:if>

<c:choose>
   <c:when test="${requestScope.message == null}">

   </c:when>
   <c:otherwise>

   </c:otherwise>
</c:choose>
```

Packaging Tag Libraries in Web Applications

With a custom tag library the tag class must be in the classpath of the class loader. Tag classes can be located in the `WEB-INF/classes` directory or stored in a JAR file placed in the `WEB-INF/lib` directory.

A Java EE 6 application server implementation can support additional directories when determining the classpath.

The standard tag libraries are already present on any Java EE 6 system as jar files. JSTL jar files have TLD files embedded in such a way that they do not require them to be listed in the `web.xml` file.

Summary

JSP pages offer a tag-based way to present dynamic data. JSP technology offers all of the advantages of servlets, with increased readability and separation of presentation and application logic.

JSP pages can be precompiled, or translated and loaded on request if the page has been modified. Servlets and JSP pages can work together by use of the `jsp:useBean` action. JSP pages can also be extended by using custom tag libraries.

Many additional JSP topics covered are covered in SL-314-EE6, Web Component Development with Servlet and JSP Technologies. Covered in detail are:

- JavaServer Pages Standard Tag Library (JSTL)
- The Unified Expression Language (EL)
- Custom tag libraries
- Error handling in JSP pages

Module 6

Developing With JavaServer Faces™ Technology

Objectives

Upon completion of this module, you should be able to:

- Evaluate the role of JavaServer Faces (JSF) technology as a presentation mechanism
- Author JSF pages using Facelets
- Process form submissions with JSF
- Describe the use of JSF tag libraries
- Use JSF managed beans

Additional Resources

Additional resources – The following references provide additional information on the topics described in this module:

- Eric Jendrock, Debbie Carson, Ian Evans, Devika Gollapudi, Kim Haase, Chinmayee Srivathsa. “The Java EE 6 Tutorial,” [http://java.sun.com/javaee/6/docs/tutorial/doc/], accessed 1 August 2009.
- "JSR 314: JavaServer Faces, Version 2.0", [http://jcp.org/en/jsr/detail?id=314], accessed 1 August 2009.
- "JSF 2.0 Page Declaration Language: Facelets Variant", [https://javaserverfaces.dev.java.net/nonav/docs/2.0/pdldocs/facelets/], accessed 10 January 2010

JavaServer Faces (JSF)

JavaServer Faces (JSF) technology is a server-side user interface component framework for building Java™ technology-based web applications. JSF pages are authored using a text-based view description language (VDL) that describes a user interface.

- JSF uses an object oriented component based approach to create user interfaces. A library of UI components are available in JSF.
- Application developers do not create JSPs or Servlet classes. JSF is an alternative method to creating Java web-based UIs. Although JSF applications will use a MVC design, it does so without using JSP or Servlet components.
- JSF 2.0 uses a page templating technology called Facelets as the VDL. Previous versions used JSPs as the VDL. JSPs are still available to use when authoring JSF pages but the use of Facelets is preferred.
- Source files for Facelets typically end with the `.xhtml` extension. Facelets are very similar in structure to XHTML documents with the addition of additional XML namespaces, tags, and the use of the Unified Expression Language (EL).

JavaServer Faces Benefits

- Reduces the amount of browser specific knowledge required to develop web applications. The JSF reference implementation includes a HTML render kit. The render kit is responsible for generating the output received by the web browser. A render kit may generate different output based on the client type.
- Object oriented approach to UI development similar to Swing.
- Maintains state across HTTP requests. For example, a JSF application can store the state of form fields across multiple HTTP requests.
- Leverages JSP knowledge by using JSP like tags and the Unified Expression Language (EL)
- Advanced type conversion and validation. A core JSF feature is the ability to convert String data to other basic Java data types and provide input validation.
- Powerful page navigation. Controller components direct users to views by specifying pages with simple text values representing the page name (not the file name). This prevents controllers from being tightly integrated with JSF Page URLs.

- Page templating based on Java EE standards. Many other templating frameworks, such as Velocity and FreeMarker, exist that can be used to generate web UIs. JSF provides a standards based page templating mechanism.

JavaServer Faces 2.0 New Features

JSF 2.0 provides the following benefits over JSF 1.2:

- AJAX tags and framework. An AJAX tag was added to JSF 2.0 which allows partial page reloading without the need to refresh the entire page. Much work was done in JSF 2.0 to lay the foundation for additional AJAX components. Additional AJAX components are available when using other component libraries.
- An annotation based configuration model that can be used in-place of XML descriptors (`faces-config.xml`). In keeping with the trend that began with Java EE 5, annotations within class files are used in-place of a deployment descriptor where possible. While a deployment descriptor may still be used in a JSF 2.0 application it will be greatly reduced in size.
- Ability to bookmark JSF pages
- Improved error handling
- New default page templating framework (Facelets). Facelets are the recommended page templating method in JSF 2.0.
- Intelligent defaults for page navigation rules. Rules are present to connect a page name with a URL. Previously this was handled using values placed in the deployment descriptor.

JavaServer Faces Development Tools

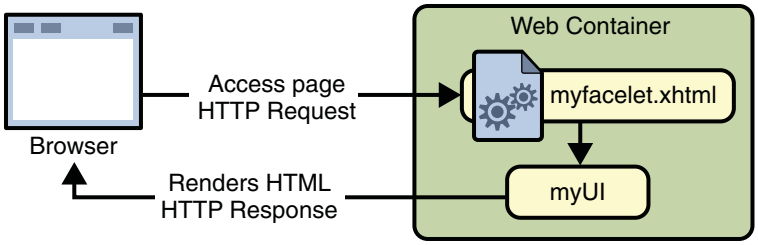
- JSF pages are authored using Facelets, a JSF-centric view technology
- Facelets are written as XML documents similar to XHTML
- Any IDE or editor that can create text or XML documents can be used to create Facelets
- Facelets typically have a `.xhtml` extension but use additional tags that are not part of the XHTML namespace
  - This may lead to false validation errors unless the editor has support for Facelets. For example, XHTML validation uses the `<body>` tag but the `<h:body>` tag is used in Facelets. While the source of the Facelet page may not be valid XHTML the output of that page may be valid XHTML.
- IDEs may have tag completion support
- Visual page designers allow pages to be designed by dragging and dropping components from a palette

Note – NetBeans previously had a visual page designer but this feature has been de-emphasized in favor of supporting tag completion.

A Server-side UI

JSF pages are rendered to a web browser but execute in the web container. A JSF UI is comprised of UI components that exist in memory and continue to exist after a request has been processed. This is why JSF is sometimes compared to AWT or Swing.

Figure 6-1 JSF UI components existing in a web container



JSF Hello World

Creating a basic “Hello World” application with JSF is a simple process. A basic JSF web application requires a developer to:

1. Create a Java EE web application
2. Create one or more Facelet pages
3. Configure the `FacesServlet` servlet in `web.xml`

Code 6-1 A `.xhtml` file placed in a Java EE web application.

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
    <h:head>
        <title>Hello World</title>
    </h:head>
    <h:body>
        <h:outputText value="A simple JSF Facelet Example"/>
    </h:body>
</html>
```

When configuring the `FacesServlet` in the deployment descriptor a `url-pattern` must be specified. This is because the web container does not treat a Facelet `.xhtml` file any different than a traditional XHTML file. The `FacesServlet` will receive the HTTP request and cause the JSF implementation to translate and execute the Facelet page. Two url-patterns are commonly used:

- `/faces/*`
- `*.xhtml`

Code 6-2 Configuring the `FacesServlet` servlet in the `web.xml` deployment descriptor.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app ... >
    <servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>*.xhtml</url-pattern>
    </servlet-mapping>
    <welcome-file-list>
        <welcome-file>index.xhtml</welcome-file>
    </welcome-file-list>
</web-app>
```

JSF Implementations and Component Libraries

A JSF implementation is included in Java EE application servers and it can be added to a servlet container such as Tomcat.

- The official JSF 2.0 reference implementation is named “Mojarra” - `https://javaserverfaces.dev.java.net/`
- Other JSF (1.2) implementations include Apache CoreJSF - `http://myfaces.apache.org/`

JSF component libraries add additional features and/or visual components to a JSF implementation. Examples include JBoss RichFaces, IceSoft IceFaces, and Apache MyFaces Trinidad.

These JSF component libraries will typically included the same UI components as the reference implementation but add additional functionality and new components.

JSF Page Tags

- JSF tags resemble JSP tags but act differently
- Focus is on view creation and restoration
- Views are rendered at the end of most Faces based requests
- No scriptlets or Java code in Facelets. Tags are all that is needed for views if you are an application developer.
- Any Java code will be in a controller or model class.
- Tag libraries are enabled in a page with a namespace declaration:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
```

JSF Tag Libraries

Prefix	Description	Namespace URI
h	JSF HTML tags. Very similar to many HTML tags. Use whenever available.	http://java.sun.com/jsf/html
f	Core Faces tags. Primarily modifies or enhances the behavior of the “h” tags.	http://java.sun.com/jsf/core
ui	Facelet Templating tags.	http://java.sun.com/jsf/facelets
c	JSTL core subset. Use the “ui” tag library if possible.	http://java.sun.com/jsp/jstl/core
fn	JSTL functions. Use the Unified EL if possible.	http://java.sun.com/jsp/jstl/functions

Managed Beans

JSF applications make use of JavaBean classes and an Inversion of Control (IoC) framework referred to as managed beans.

- JavaBean classes have a zero-arg constructor and use getters and setters to expose properties.
  - `http://java.sun.com/products/javabeans/`
- A JSF application will instantiate JavaBean objects and access their properties if used in an EL expression.
- A common design is to have a single managed bean per page view or form. A managed bean used in this fashion is referred to as a backing bean.
- Managed beans are used to hold form values and methods that are executed by form submission.

Note – Java EE 6 includes *JSR 299: Contexts and Dependency Injection for the JavaTM EE platform* (CDI). CDI can be used in place of the traditional JSF managed beans facility. While some of the annotations are different when using CDI, it does not significantly change the design of a JSF application. For more on CDI see: `http://jcp.org/en/jsr/detail?id=299`

The Unified Expression Language (EL)

The Unified Expression Language (EL) is “unified” because both JSP and JSF use the same Expression Language. It is used slightly differently in JSF pages than in JSP pages. JSF pages:

- Typically use a # symbol instead of a $. For example: `#{param.field1}`. The # indicates deferred evaluation and the $ indicates immediate evaluation.
- Have several predefined objects such as `param`
- Unlike JSPs, can use the EL to read and write bean properties. This means setter methods can be called. Other methods, known as action methods, can be invoked using EL within a JSF page.

JSF Forms

The JSF HTML tags from the http://java.sun.com/jsf/html namespace will be used to create forms. When creating a JSF form:

- Use the JSF HTML tag library. Refer to the Page Description Language (PDL) docs for a complete list of form related tags.
- The form will be submitted back to the URL that rendered the form instead of submitting to a Servlet or other URL.
- The JSF implementation parses form data and makes it available to managed beans
- Managed bean properties are associated with form fields though the Expression Language

Code 6-3 A basic JSF form example

```
<h:form>
    <h:outputLabel for="guess" value="Your Guess: " />
    <h:inputText id="guess" value="#{gameBean.guess}" />
    <h:commandButton action="#{gameBean.checkGuess}" value="Submit" />
</h:form>
```

Managed Bean Configuration

After creating a JavaBean it must be configured to act as a managed bean. Configuration can be done in two ways:

- Annotation based configuration (JSF 2.0)

```
@ManagedBean
public class NumberGameBean {...}
```

- XML based configuration (`faces-config.xml`)

```
<managed-bean>
  <managed-bean-name>...</managed-bean-name>
  <managed-bean-class>...</managed-bean-class>
</managed-bean>
```

Managed Bean Names

- A managed bean has a name that can be used in an EL expression.
  `#{gameBean.guess}`
- The annotation based configuration derives the default name from the classname starting with a lowercase letter.
- An EL name can be specified with a name attribute.

```
@ManagedBean(name="gameBean")
public class NumberGameBean {...}
```

Managed Beans and Forms

Managed Beans are often used to maintain form data and handle form submission.

- Input form elements are associated with bean properties through the use of the EL.
- Form input elements have a value attribute
  `value="#{gameBean.guess}"`
- JSF uses the setters and getters for bean properties.
- When displaying a form the getter will be used.
- When submitting a form the setter will be used.
- Form submission is associated with a Java method using an action attribute

```
<h:commandButton action="#{gameBean.checkGuess}" value="Submit" />
```

Managed Beans and Text Output

Text content, including traditional HTML tags, may be placed within a JSF page. This text content will be passed, unmodified, to the web browser. The `<h:outputText>` tag can also be used to produce text output.

```
<h:outputText value="A simple JSF Facelet Example"/>
```

The EL can be used any place in a JSF page to read the value of a bean property and display it.

```
Guess the number, 1-#{gameBean.max}
```

A benefit of using the `<h:outputText>` tag with an EL expression in the value is that any value returned by the bean can be escaped. This behavior can be controlled with the `escape="true"` attribute.

Managed Bean Lifecycle and Scope

Managed beans are instantiated when they are first used in an EL expression on a page. By default managed beans are discarded after each request. The lifetime of a bean can be extended by configuration, for example:

```
@ManagedBean(name="gameBean")
@SessionScoped
public class NumberGameBean {...}
```

Commonly used scopes are:

- `@SessionScoped`
- `@RequestScoped`
- `@ApplicationScoped`

JSF Page Navigation

Submitting a form often will require that the user be redirected to a different page. A submit button is rendered using the `<h:commandButton>` tag.

Form submission is tied to a method in managed bean using the action attribute of the `<h:commandButton>` tag. When the submit button is pressed the method specified using EL will be invoked. This action method returns a String, the page name to display. No extensions, such as `.xhtml`, are used.

Code 6-4 A `<h:commandButton>` tag example

```
<h:form>
    <h:outputLabel for="guess" value="Your Guess: " />
    <h:inputText id="guess" value="#{gameBean.guess}" />
    <h:commandButton action="#{gameBean.checkGuess}" value="Submit" />
</h:form>
```

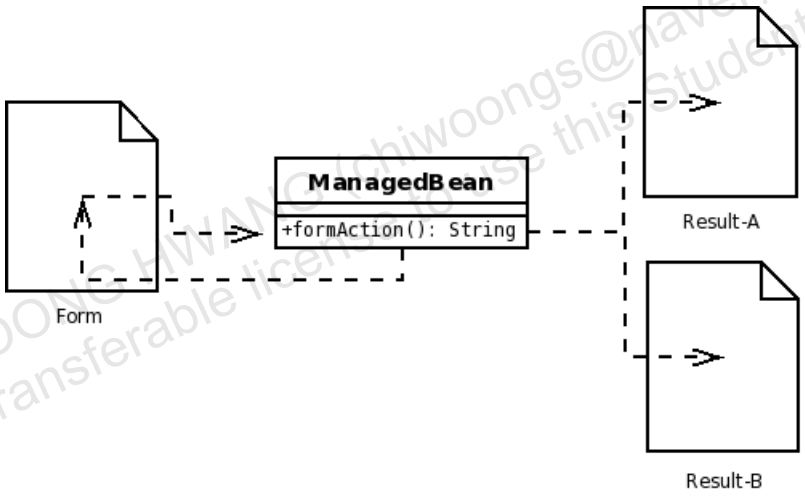


Figure 6-2 Page navigation with a managed bean

A `null` return value will re-display the page the contains the form.

Validation

Form input elements can use the `required` attribute to ensure that a field has been filled out.

```
<h:inputText id="guess" value="#{gameBean.guess}" required="true" />
```

JSF provides tags to validate form data. String length validation can be performed with the `<f:validateLength>` tag.

```
<h:inputText id="username" value="#{logonBean.username}">
    <f:validateLength minimum="6" />
</h:inputText>
```

The `<f:validateLongRange>` and `<f:validateDoubleRange>` tags validate numeric ranges.

```
<h:inputText id="guess" value="#{gameBean.guess}" required="true" />
    <f:validateLongRange minimum="1" maximum="#{gameBean.max}" />
</h:inputText>
```

Additional validation tags such as `<f:validateRegex>` are available, refer to the Page Description Language (PDL) docs for a complete listing.

Input Conversion

Form fields are submitted as String data and may need to be converted before being stored in a managed bean property. Basic conversion, such as `String` to `int`, takes place automatically.

Conversion can be customized with built-in convertors. The `<f:convertNumber>` tag can be used to display numeric values as currency amounts.

```
<h:inputText value="#{myBean.salary}">
<f:convertNumber type="currency"/>
</h:inputText>
```

Validation and Conversion

Both validation and conversion tags require the Core Faces tag library.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
```

Validation and conversion tags can be used at the same time for a form element.

```
<h:inputText value="#{myBean.salary}">
    <f:validateDoubleRange minimum="1" maximum="100"/>
    <f:convertNumber type="currency"/>
</h:inputText>
```

Validation and Conversion Errors

If a validator or convertor fails then the submitted value will NOT be stored in the managed bean. The form page will be re-displayed so the error can be corrected.

By default there is no visual indication that validation or conversion has failed when the form is re-displayed. To include an unordered list of error messages add the <h:messages/> tag to the form page.

```
j_id1053692265_3ece1132:j_id1053692265_3ece11eb: 'one' could not be
understood as a currency value.
```

An unfriendly label is used by default. Use the label attribute of form field tags to make the error message user friendly.

```
<h:inputText value="#{myBean.salary}" label="Salary" >
    <f:validateDoubleRange minimum="1" maximum="100"/>
    <f:convertNumber type="currency"/>
</h:inputText>
```

The list item rendered by the <h:messages/> tag displays:

```
* Salary: 'one' could not be understood as a currency value.
```

Single error messages can be displayed with the <h:message> tag.

```
<h:message for="salary"/>
```

The for attribute of the <h:message/> tag uses the same value as the id attribute of the <h:inputText> tag.

```
<h:inputText id="salary" value="#{myBean.salary}" label="Salary" >
    <f:validateDoubleRange minimum="1" maximum="100"/>
    <f:convertNumber type="currency"/>
</h:inputText>
```

The error message is rendered as plain text, not as a list item.

```
Salary: 'one' could not be understood as a currency value.
```

Error Messages

The <h:messages/> and <h:message/> tags display the generic error messages. To display custom messages use one of the following attributes on form field tags:

- requiredMessasge="..."
- converterMessage="..."
- validatorMessage="..."

```
<h:inputText value="#{myBean.salary}" converterMessage="Enter a currency
amount">
    <f:validateDoubleRange minimum="1" maximum="100"/>
    <f:convertNumber type="currency"/>
</h:inputText>
```

Internationalization

Any text on a page, including error messages, may need to be localized for the current user. JSF allow localized messages bundles to be easily used with a page. To use i18n in JSF:

- Create java.util.Properties files for each language
- File names should include the supported locale such as messages.properties (the default file), messages_en_US.properties, messages_de.properties
- Message files are placed in the CLASSPATH (in a Java package directory within your IDE)
- Configure a JSF resource bundle
- Use the EL in JSF pages to read a localized message

Internationalization Demo

A `messages.properties` file might contain:

```
greeting=Howdy
```

Within a JSF page the localized greeting message can be displayed using the EL.

```
#{msgs.greeting}
```

Localized messages are also used for conversion and validation error messages.

The `faces-config.xml` file is used to configure a JSF application and is placed within the `WEB-INF` folder of a web application.

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd"
   version="2.0">
   <application>
      <resource-bundle>
         <base-name>com.example.messages</base-name>
         <var>msgs</var>
      </resource-bundle>
   </application>
</faces-config>
```

The example above makes the `com/example/messages.properties` file available in all JSF pages with a EL variable of `msgs`.

The message displayed is determined by the available resource bundles and by the users locale. By default the users local is determined by the `Accept-Language` HTTP header.

To override a users local use the <f:view> tag as shown:

```
<html ...>
    <f:view locale="en_US">
       <h:head>...</h:head>
       <h:body>...</h:body>
    </f:view>
</html>
```

An EL expression can be used as the value of the locale attribute.

DataTable

A JSF DataTable is used instead of a <c:forEach> tag to render HTML tables.

```
<h:dataTable value="#{gameBean.guesses}" var="guess">
    <h:column>
       <f:facet name="header">Guesses</f:facet>
       #{guess}
    </h:column>
</h:dataTable>
```

The <f:facet> tag is optional and only required if a <th> is desired in the resulting output.

Summary

The following topics were presented in this module:

- JSF technology as a presentation mechanism
- The creation of JSF pages (Facelets)
- JSF page navigation
- Conversion and validation
- Internationalization (i18n)

Module 7

EJB™ Component Model

Objectives

Upon completion of this module, you should be able to:

- Describe the role of EJB components in a Java EE application
- Describe the EJB component model
- Identify the proper terminology to use when discussing EJB components and their elements

Additional Resources

Additional resources – The following references provide additional information on the topics described in this module:

- "Java Platform, Enterprise Edition 6 (Java EE 6) Specification", [http://jcp.org/en/jsr/detail?id=316], accessed 1 August 2009.
- "JSR 318: Enterprise JavaBeans, Version 3.1", [http://www.jcp.org/en/jsr/detail?id=318], accessed 1 August 2009.

Role of EJB Components in a Java EE Application

A Java EE application uses a suite of components managed by the EJB container to provide the business and messaging capabilities that are required by an enterprise business application. An application developer can use EJB components to create distributed business applications that are scalable, transactional, multi-user, and secure.

EJB Component Types

Figure 7-1 illustrates the components that are hosted by the EJB container.

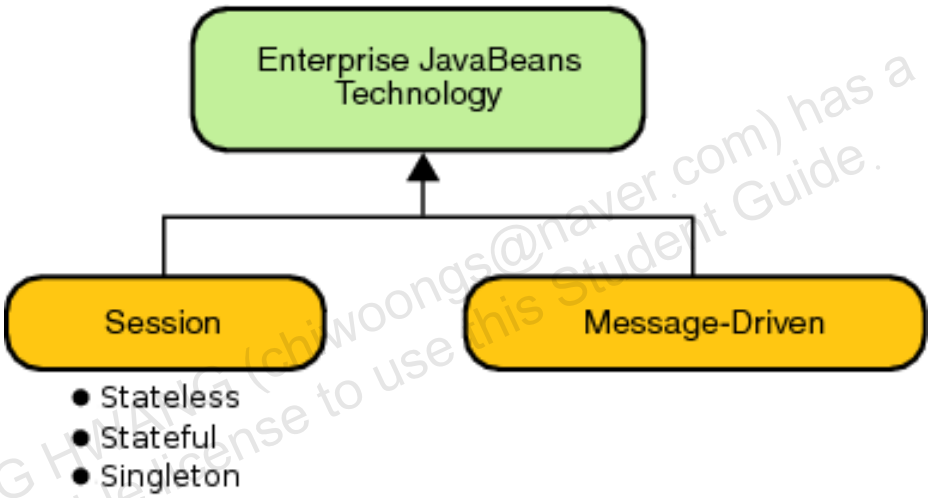


Figure 7-1 EJB Component Types

There are two basic types of Java EE components that must be located in an EJB container:

- Session EJB components – A session EJB component provides a service to clients by implementing business logic that runs in an EJB container. A session EJB component might access the database and might also coordinate transactional work that involves multiple entity classes.
- Message-Driven Beans – A message-driven bean is an asynchronous message consumer that can execute server-side business logic based on messages received from a variety of sources, such as legacy systems and Java EE clients. Although message-driven beans are located an EJB container, they have different operational characteristics from session components.

Note – Previous versions of the Enterprise JavaBeans specification included an entity EJB component. Entity EJB components are still supported for backwards compatibility, but have been replaced by a non-EJB entity component outlined in the new Java Persistence API specification.

EJB Tiers

Modern practice favors the division of enterprise components into tiers. The Java EE specification does not mandate this division. It is more of a design pattern. There are generally two major enterprise tiers that might be further subdivided depending on the application design:

- The *services* tier, which consists mostly of session beans and message-driven beans
- The *EIS-facing* or *object-relational mapping* tier, which consists mostly of entity classes and their supporting classes

Note – The term layer is sometimes used in place of tier. In this module the term tier does not designate a physical separation, instead it notes a logical partitioning of functionality.

An EJB application might have a large number of clients, and the clients might, in principle, interact with both session bean services and entity components. However, in a multi-tiered EJB application, it is considered bad design to have clients, that are not themselves EJB components, interact directly with the database or entity components.

Session beans are common clients for entity components. It is common design practice to use a session bean as a facade for a number of entity components, with most of the computational logic encapsulated in the session bean.

There are two main reasons to use a session bean as a facade for a number of entity components:

- Entity components in an EJB container do not provide remote access to clients.

  The use of course-grained session bean interfaces minimize network overhead and allow developers to leverage container managed transactions.
- To simplify the clients.

  Clients of an EJB application expect to see services, which are provided by session beans, not data, which is provided by entity beans. Clients must contain transaction management code when directly using entity beans.

Note – This design practice is also called the *Session Facade pattern*, which is presented in more depth in SL-355, *Business Component Development with Enterprise JavaBeans™ Technology* and in SL-500, *Java EE™ Patterns*.

Figure 7-2 shows two session beans, `AutomaticTeller` and `BranchTeller` that expose the business logic of a banking application.

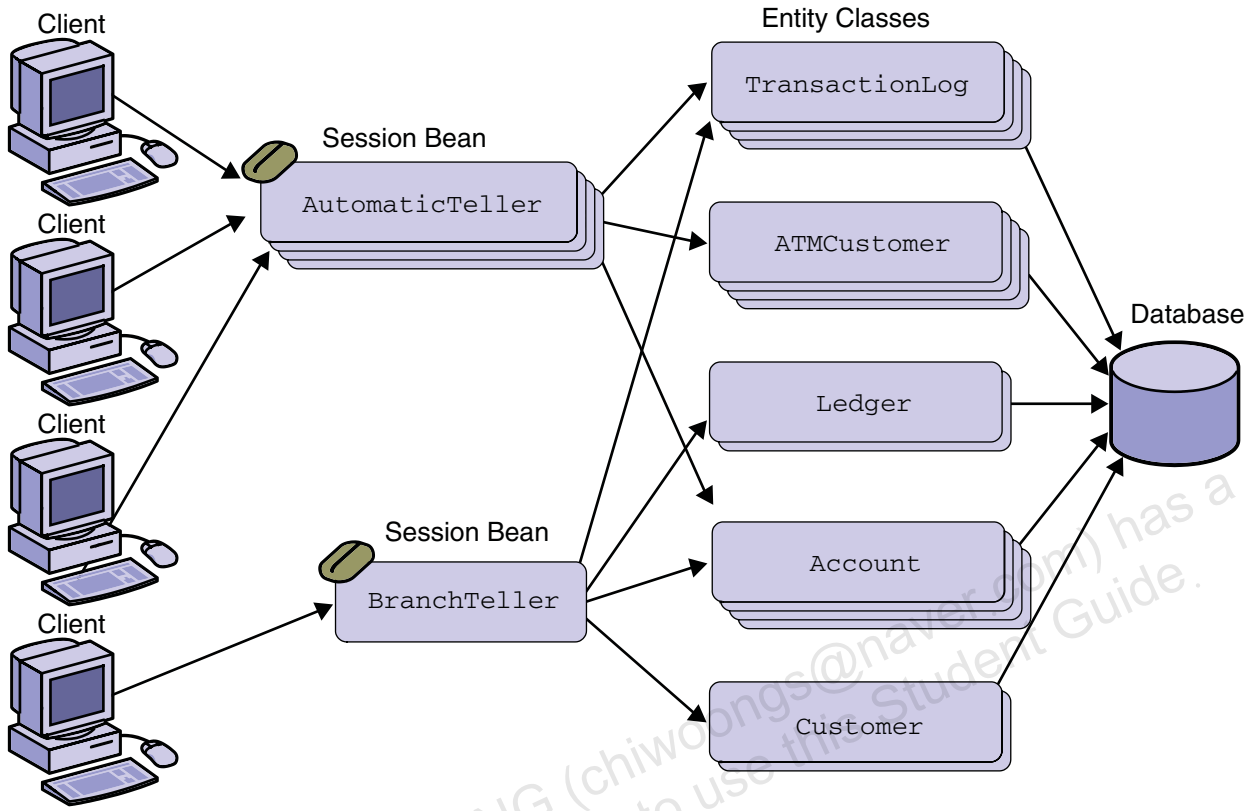


Figure 7-2 EJB Application Tiers

In Figure 7-2, bank clients make deposits and withdrawals using the functionality that is exposed by the `AutomaticTeller` session bean. Tellers create new accounts and perform other banking functions using the `BranchTeller` session bean. The application session beans connect with the back-end data store using a suite of entity classes that model the business data.

Each entity class represents a different type of business entity. For example, an entity class can represent transaction logs, ATM customers, ledgers, accounts, and customers. At the end of a transaction, the data that is contained in entity components are typically written to the database.

EJB Method Types

There are two basic categories of methods that are present in session EJB components:

- Life-cycle or callback methods – These methods are called by the container at various points in the life cycle of an EJB component or associated implementation class. An EJB component developer can implement these methods depending on the functionality required from the EJB component.
- Business methods – Typically defined by a business interface, the bean class implements business methods that execute the logic that EJB components require.

Important EJB Component Interfaces

The EJB component model provides some basic functionality using a few base interfaces that are common to all EJB component types. These interfaces include:

- The `EJBContext` interface – This interface provides access to the container-provided runtime context of an enterprise bean instance. The `SessionContext` and `MessageDrivenContext` interfaces extend the `EJBContext` interface and provide additional methods that are specific to the enterprise bean type. There are several security, transaction-related, and lookup methods on the `EJBContext` interface that a component developer might find useful.

Analysis of the EJB Component Model

The EJB component model has the following characteristics:

- The model uses containers to encapsulate the component.
- The container provides proxies that allow clients limited and controlled access to the EJB component.
- The proxies implement interfaces that are provided by the EJB component developer.
- Clients make method calls on these interfaces.

Each of these characteristics is presented in the following sections.

Role of the EJB Container

The EJB container provides a number of important services to EJB components, as shown in Figure 7-3.

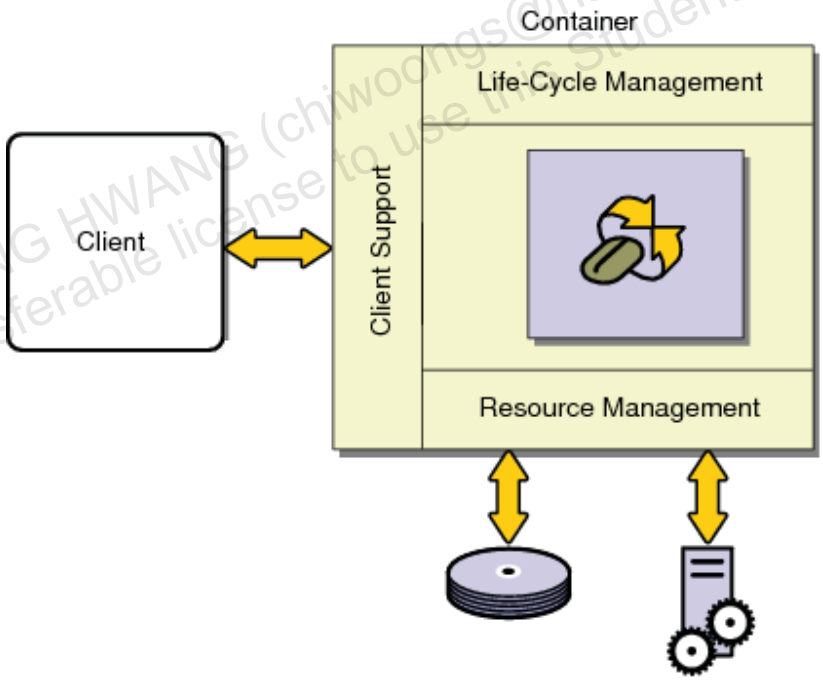


Figure 7-3 Role of the EJB Container in the EJB Component Model

The EJB container provides the following functions:

- The container encapsulates access to external resources, such as databases and legacy systems. In doing so, the container provides connection pooling services to allow these resources to be used in the most efficient way.

- The container manages the life cycles of instances of the EJB component's implementation class. The way in which this management is done varies according to the type of component. For example, clients of session bean components can request an EJB proxy and the container creates instances as needed.
- The container isolates the class that provides the implementation from its clients. Clients have to call through the `EJBOjbect` in the container to invoke methods on the implementation. This has important benefits, including the ability to implement security and transaction management declaratively, rather than in the code.
- The container provides timer services, and can invoke methods at certain times. This service allows the EJB component to perform scheduled tasks.
- For message-driven beans, the container monitors a message queue on behalf of the EJB component.

Embedded EJB Container

Java EE 6 adds support for an embedded EJB Container. A Java application may use the embedded EJB container in place of the application server hosted EJB container. Example application types that may use the embedded container included Java SE desktop applications and Java EE web applications.

An Embedded EJB container supports the EJB Lite subset of the EJB API. Vendors may optionally provide the full EJB API in an embedded EJB container.

Code 7-1 Embedded EJB Container usage

```
EJBContainer ec = javax.ejb.embeddable.EJBContainer.createEJBContainer();
Context ctx = ec.getContext();
FooLocal foo = (FooLocal)ctx.lookup("java:global/foo/FooBean");
// use EJB
ec.close();
```

EJB Objects and Proxies

When an EJB component client requests a session component, the container returns to the client a reference to the EJB component's *EJB object*, as shown in Figure 7-4.

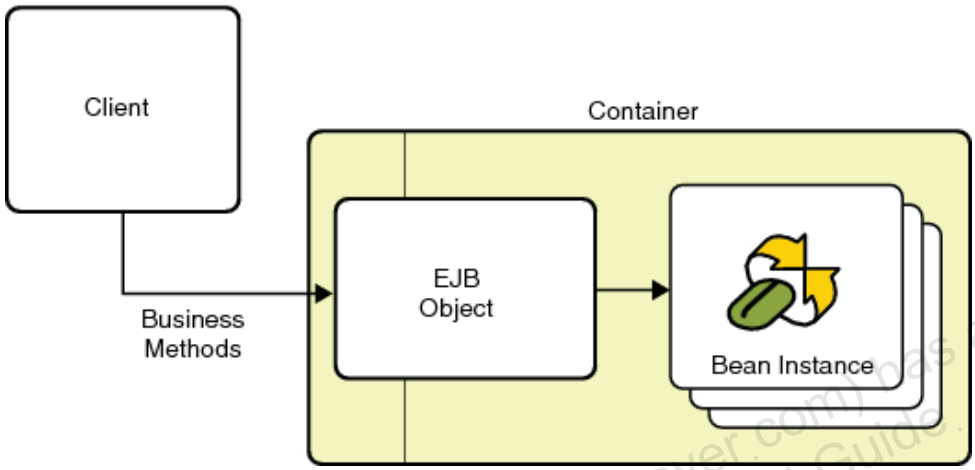


Figure 7-4 EJB Object as a Proxy for the EJB Component Model Implementation

The EJB object class is generated automatically by the server based on the definition of the *component or business interface*. All subsequent client interactions are through the EJB object.

Local and Distributed Client Views

Because the Java EE model enforces a strict separation between interface and implementation, the client interacts with these objects by their interfaces. Consequently, the business interface serves the following three purposes:

- It provides an access point and type for clients.
- It specifies to the server how to construct the EJB object.
- It serves as a framework for the behavior of the EJB component.

You can provide a session EJB component with any combination of the following four types of client views:

- Local interfaces – The local interfaces are collectively known as the *local client view.*

- Distributed interfaces – Together, the distributed interfaces form the *distributed* or *remote client view.*
- Web services interfaces – The web services interfaces for stateless session beans are known as the *Web Services view.*
- No-interface view – An EJB 3.1 variation of the local client view. The public methods of the bean class specify the EJB component's client API. No business interface is written.

Different interfaces are required because local and distributed interactions have different calling semantics.

Elements of a Distributable EJB Component

A distributable, remotely accessible, EJB component has the following elements:

- Remote component interface

  The remote component interface provides access to the business methods of the EJB component. The remote component interface is implemented by a stub or proxy on the client side, and the *EJB object* on the server side.

  For session EJB components, an EJB client uses dependency injection or JNDI lookups to obtain a reference to the remote component interface. The EJB object is also called the *component object*.

- Implementation class

  This class is marked with the `javax.ejb.Stateless` or `javax.ejb.Stateful` annotation. The implementation class is often called the *bean class*, *implementation instance*, *EJB Instance,* or *EJB class*. Clients do not have direct access to instances of the implementation class. Implementation instances are always accessed through remote or local interfaces.

Client View of a Distributable EJB Component

Because a client interacts with the EJB component through interfaces, the client view of an EJB component is not that different from the client view of an ordinary Java object. The process of obtaining a reference to an object was introduced in “Use of the Java Naming and Directory Interface™ (JNDI) API in the Java EE Component Model” on page 2-11. Code 7-2 illustrates how a client connects to and uses a session EJB component named `bankMgr`.

Code 7-2 Client View of a Distributed Component

```
1   BankMgr bankMgr;
2   try {
3      InitialContext ic = new InitialContext();
4      bankMgr = (BankMgr) ic.lookup ("java:comp/env/ejb/BankMgr");
5   } catch (...) {
6      // Handle exceptions
7   }
```

In the preceding code example, there is no indication about whether the client is using the local view or the distributed view of the EJB component. In fact, the client code is essentially identical in the local or distributed view of the EJB component. However, it is conventional to name the interfaces differently for the local view. So, if this were a local client view interaction, the interface would conventionally be called `BankMgrLocal`.

EJB Life Cycle

An EJB component has a life cycle that is controlled by the EJB container based on the functionality that is requested by the client and the operational requirements of the system. Each type of EJB component has specific life-cycle operations and characteristics. Understanding the life cycle of the various EJB component types might allow a component developer to optimize the operation of the bean.

EJB Timer Service

All EJB component types can use the EJB Timer Service, a feature that was added as of the Java EE platform 1.4 and updated for Java EE 6. The EJB timer service is a container-managed service that enables the bean developer to register enterprise beans for timer callbacks. These timer callbacks enable enterprise applications to model workflow-type business processes that depend on notifications that certain time-related events have occurred.

The container provides a reliable and transactional notification service for timed events. Timer events can be scheduled to occur in one of four ways:

- At a specific time
- After a specific elapsed duration
- At specific recurring intervals
- On calendar schedules similar to UNIX cron (new to Java EE 6)
- On all EJB types except statefull session beans

Events can be scheduled programmatically or they can be created automatically with deployment descriptors and bean annotations.

The bean annotation method uses `@Schedule` annotation. This annotation use a calendar-based syntax that is modeled after the UNIX cron facility.

Code 7-3 A `@Schedule` annotation example

```
@Schedule(hour="1", dayOfMonth="1")
public void generateMonthlyAccountStatements() { ... }
```

Timers can be created for stateless session beans, singleton session beans, message-driven beans, and 2.1 entity beans. Timers cannot be created for stateful session beans.

`@Schedule` Syntax

The `@Schedule` annotation is placed on a method in an EJB such as a stateless session bean. There is no required name for the method. The `@Schedule` annotation accepts the following attributes:

- `second`
- `minute`
- `hour`
- `dayOfMonth`
- `month`
- `dayOfWeek`
- `year`

The `second`, `minute`, and `hour` attributes have a default value of "0". The `dayOfMonth`, `month`, `dayOfWeek`, and `year` attributes have a default value of "*".

`@Schedule` Examples

The following examples are just some of the examples that can be found in the EJB 3.1 specification.

Code 7-4 Every Monday at Midnight

```
@Schedule(dayOfWeek="Mon")
```

Code 7-5 Every Weekday morning at 3:15

```
@Schedule(minute="15", hour="3", dayOfWeek="Mon-Fri")
```

Code 7-6 Every minute of every hour of every day

```
@Schedule(minute="*", hour="*")
```

Code 7-7 Every five minutes within the hour

```
@Schedule(minute="*/5", hour="*")
```

For more information, refer to the SL-355, *Business Component Development With Enterprise JavaBeans™ Technology* course or to the Enterprise JavaBeans Specification, version 3.1.

Calling EJB Components From Servlets

In principle, servlets can make any number of method calls on session beans. However, there are a few points that you should keep in mind to avoid inefficiencies in the interaction between servlets and EJB components:

- To reduce overhead, servlets should generally make a few coarse-grained method calls, rather than many fine-grained method calls.
- References for EJB components could be located at initialization time to avoid repeated lookups.

Initializing a Reference to an EJB

The snippet shown in Code 7-8 illustrates how a servlet can locate the EJB component at initialization time and store it in an instance variable for future use.

Code 7-8 Initializing a Reference to a Stateless Remote Session Bean

```
1   private BankMgr bankMgr = null;
2
3   public void init() {
4     try {
5       InitialContext ic = new InitialContext();
6
7       bankMgr = (BankMgr)ic.lookup ("java:comp/env/ejb/BankMgr");
8
9     } catch (Exception e) {
10      throw new ServletException(e);
11    }
12  }
```

You should note the following about Code 7-8 on page 7-15:

- The `InitialContext` object is created using a default constructor and there are no JNDI API properties.
- The servlet uses a `java:comp/env` name to locate the EJB component in the web module's local namespace.
- The initialization can throw exceptions.

  Your code should handle these exceptions in such a way that subsequent calls to the `service` method fail gracefully. In the code example, if lookup fails, the `bankMgr` reference remains null. It is easy to test for this condition in the service method.

Using Annotations to Obtain an EJB Reference to a Managed Component

Annotations that cause dependency injection of EJB components can only be used in a managed component. A managed component is an enterprise component, such as a servlet or session bean that has its life cycle managed by a container. The snippet shown in Code 7-9 illustrates how a servlet can locate the EJB component at initialization time using an EJB annotation.

Code 7-9 Using Annotations to Obtain a Reference to an EJB

```
1   @EJB private BankMgr bankMgr;
```

You should note the following about Code 7-9:

- A JNDI lookup or context is not required.
- The container assigns a value to the `bankMgr` reference variable when the servlet is instantiated.
- The initialization does not throw exceptions.

When you store a reference to a stateless session EJB component in an instance variable, you do not compromise thread safety for two reasons. First, the variable is only written once when the servlet is initialized. Second, calls on the business methods of a stateless session EJB component are guaranteed to be thread-safe.

There are circumstances in which it is safe to store a reference to the EJB component itself in an instance variable, but this approach is not advised unless you are absolutely familiar with the EJB component-thread model. Concurrent access to a stateful session bean will be serialized in Java EE 6, previous versions could have possibly raised an `ConcurrentAccessException`.

EJB Lite

Many EJB applications will not make use of the full EJB API. In order to make the most frequently used portions of the EJB API available to more applications a subset of the EJB API has been defined as EJB 3.1 Lite. EJB Lite is a required feature of embedded EJB containers and the Java EE web profile.

EJB 3.1 Lite supports:

- Stateless, Stateful, and Singleton Session Bean components
- Local and no-interface view only
- Synchronous method invocations only
- Container-managed transactions and Bean-managed transactions
- Declarative and programmatic Security
- Interceptors

Notably not supported are Message-driven beans, the EJB Timer service, and remote EJBs.

EJB Components Before the Java EE 5 Platform

There are a variety of terms that are used to talk about EJB components. Before the EJB 3.0 specification, EJB components were more complex and many companies might wish to convert Java EE projects to use EJB 3.0 or 3.1 to reduce the complexity of applications. Before the existence of local client views, introduced in the EJB 2.0 specification, the component interface was often referred to as the remote interface, or as the EJB object. With the introduction of local client views, and associated new terminology, the term *remote* has a more specific meaning when referring to interfaces.

Unfortunately, old terminology does not always fade from use as fast as new technology is introduced. You might hear colleagues referring to a remote interface, when, in fact, they mean to be referring to the component interface, regardless of its local or remote client view. Table 7-1 provides a list of EJB component terminology, along with their descriptions and equivalents.

Table 7-1 EJB Component Terminology

Version	Term	Description
1.1	Remote client view	Includes both the home and remote interface
	Home interface	Extends the `EJBHome` interface and provides a n
	Remote interface	Extends the `EJBObject` interface and provides an EJB object

Table 7-1 EJB Component Terminology (Continued)

Version	Term	Description
2.0/2.1	Remote client view	Includes both the remote home interface and the remote interface
	Local client view	Includes both the local home interface and the local interface
	Home interface	Generic term for the factory interface for an EJB component, regardless of whether it is local or remote
	Local home interface	Extends the `EJBLocalHome` interface and provides a local home object
	Remote home interface	Extends the `EJBHome` interface and provides a remote home object
	Component interface	Generic term for the business method interface, regardless of whether it is local or remote
	Local component interface	Extends the `EJBLocalObject` interface and provides a local EJB object
	Remote component interface	Extends the `EJBObject` interface and provides a remote EJB object
3.0/3.1	Business Interface	Generic term for the business method interface, regardless of whether it is local or remote
	Local Interface	Marked with Local annotation (default) and provides a local EJB object
	Remote Interface	Marked with Remote annotation (default) and provides a remote EJB object

Summary

A Java EE application uses a suite of components that is managed by the EJB container to provide the business functionality that is required by an enterprise business application. An application developer can create distributed business applications that are scalable, transactional, multi-user, and secure using EJB components.

The two basic types of Java EE components that are located in the EJB container are session EJB components and message-driven beans.

Modern practice favors the division of enterprise components into tiers. There are generally two major tiers that might be further subdivided into the following tiers, depending on the application design:

- *Services* tier, which is typically implemented by one or more session EJB components based on the Session Facade design pattern
- *EIS-facing* or *object-relational mapping* tier, which is typically implemented by non-EJB entity components

Two general classifications of methods found on an EJB component are life-cycle methods and business methods.

The EJB component model has the following characteristics:

- The model uses containers to encapsulate the component.
- The container provides proxies that allow clients limited, controlled access to the EJB component.
- The proxies implement interfaces that are provided by the EJB component developer.
- Clients make method calls on these interfaces.

The terminology used to describe the elements of an EJB component has changed as the EJB specification has evolved.

Module 8

Implementing EJB 3.1 Session Beans

Objectives

Upon completion of this module, you should be able to:

- Compare stateless and stateful behavior
- Describe the operational characteristics of a stateless session bean
- Describe the operational characteristics of a stateful session bean
- Create session beans
- Package and deploy session beans
- Create a session bean client

Additional Resources

Additional resources – The following reference provides additional information on the topics described in this module:

- "Java Platform, Enterprise Edition 6 (Java EE 6) Specification", [http://jcp.org/en/jsr/detail?id=316], accessed 1 August 2009.
- "JSR 318: Enterprise JavaBeans, Version 3.1", [http://www.jcp.org/en/jsr/detail?id=318], accessed 1 August 2009.

Comparison of Stateless and Stateful Behavior

Most Java EE applications that have user interfaces require the retention of state data. Examples of data that must be retained are multi-page forms and shopping carts. State can be stored in any number of places including:

- User tier – State can be stored in objects in Swing-based applications or in cookies with web-based applications. This is typically avoided because of security and performance reasons. Large amounts of data would have to frequently be transferred between the client and server tiers.
- Web tier – Web clients can have their state stored on the web tier in `javax.servlet.http.HttpSession` objects. If the web and EJB business tiers are running in the same application server, there is not a drastic performance penalty if local business components are used. State management should be done in a centralized business tier if non-web clients are used for reusability.
- Database tier – Sometimes used to achieve increased retention time of state. Can provide increased failover capabilities when a vendor's Java EE application server lacks or has not been configured with advanced clustering and failover features.
- Business or EJB tier - Session EJBs can hold client state. Because both web and Swing UI applications can store state in the EJB tier, this allows most state management code to be written only once. This is typically easier and faster than storing state in a database.

In a Java EE 6 application, the business tier contains session beans. Session beans can be configured as stateless or stateful. The required statefulness of a bean depends on the type of business function it performs:

- In a stateless client-service interaction, no client-specific information is maintained beyond the duration of a single method invocation.
- Stateful services require that information obtained during one method invocation be available during subsequent method calls:
  - Shopping carts
  - Multi-page data entry
  - Online banking

Stateless Session Bean Operational Characteristics

A stateless session bean has the following characteristics:

- The bean does not retain client-specific information.
- A client might not get the same session bean instance.
- A large number of client requests can be handled by the same session bean. Each request is routed to a different bean instance by the container. This has a profound impact on performance.

Memory usage is decreased and performance increased when using stateless session beans, when compared to unnecessarily configuring a session bean as stateful.

Note – A common misconception is that stateless session beans cannot have instance variables. It is valid for stateless session beans to have instance variables; however, clients might not get the same bean on every method call, so from a client's perspective, the value of the variables can appear to change.

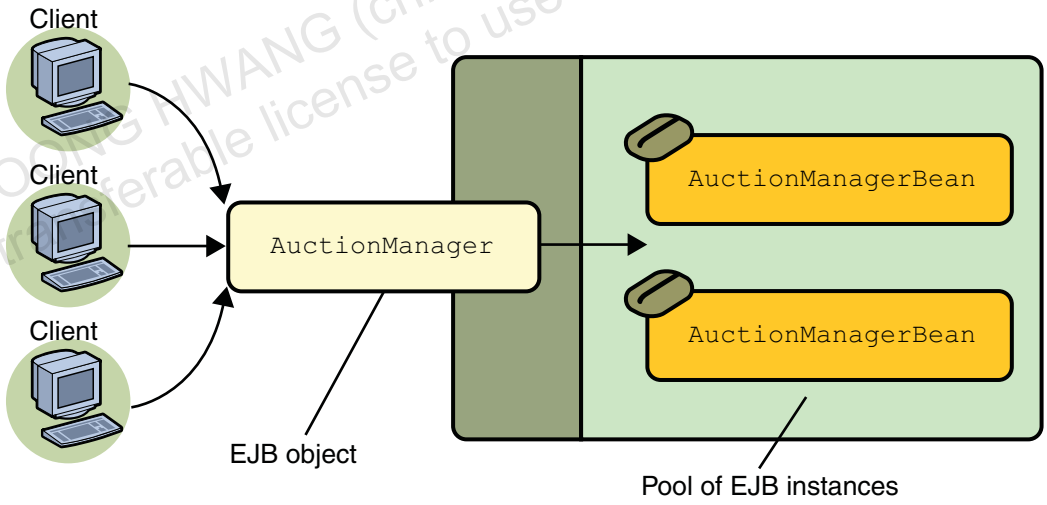


Figure 8-1 Stateless Session Bean Cardinality

Stateful Session Bean Operational Characteristics

A stateful session bean has the following characteristics:

- The bean belongs to a particular client for an entire conversation or session.
- The client connection exists until the client removes the bean or the session times out. Session bean time outs are typically based on a vendor-specific inactivity timeout threshold.
- The container maintains a separate EJB object and EJB instance for each client. Stateful session beans require more memory per client than stateless session beans.

Note – Another common misconception is that stateless session beans scale better than stateful session beans. There is always a cost to maintain client state and maintaining more state than is needed, or using stateful session beans when a stateless bean would be adequate, negatively impacts performance. If an application must store client state, then stateful session beans are an effective way to store the client state.

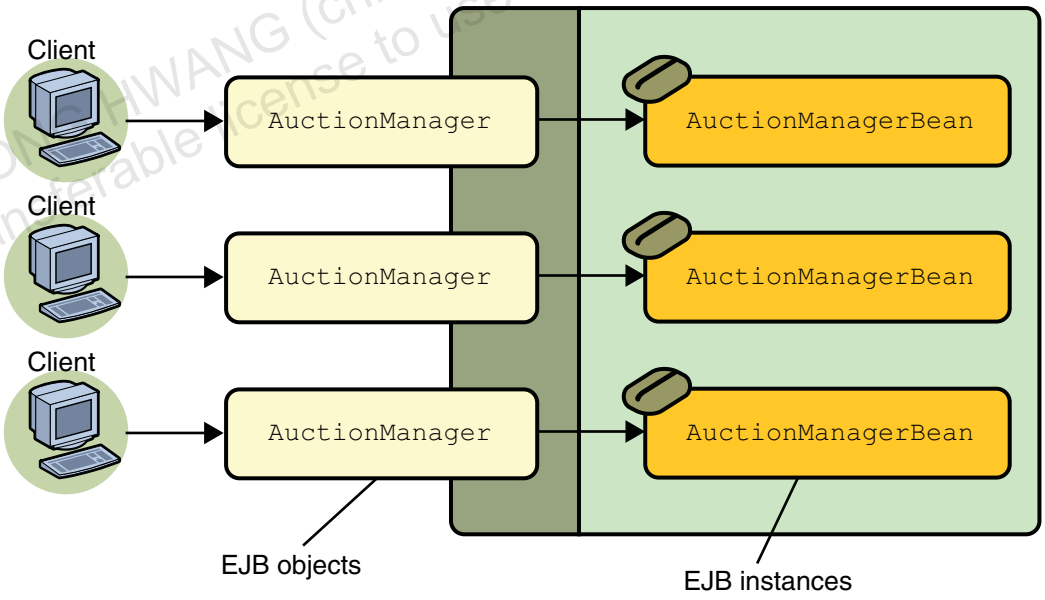


Figure 8-2 Stateful Session Bean Cardinality

Singleton Session Bean Characteristics

A singleton session bean has the following characteristics:

- The container maintains a single EJB instance for each application. If any application is distributed across multiple JVMs there will be an instance for each application per JVM.
- The bean can belong to a multiple clients simultaneously. Concurrent access is controlled using Container Managed Concurrency by default.
- The bean does not retain state across server shutdown or restart.

Unlike stateless session beans, a singleton session bean will maintain state across method calls. Additionally state is shared by all clients as all clients will access the same bean instance. Use the `javax.ejb.Singleton` annotation to create a Singleton session bean.

```
@Singleton
public class CacheBean {...}
```

The methods of a singleton session bean can be concurrently accessed by multiple clients. By default a singleton session bean's concurrent access is controlled by the container using Container Managed Concurrency. Container Managed Concurrency supports a multiple reader, single writer locking strategy. Every method of a Singleton session bean attempts to obtain a Write lock by default. Methods can be annotated with either:

- `@javax.ejb.Lock(javax.ejb.LockType.WRITE)`
- `@javax.ejb.Lock(javax.ejb.LockType.READ)`

Creating Session Beans

This section provides an overview of the essential tasks you need to perform to create a EJB 3.0 session bean. To create a session bean you are required to perform the following tasks:

- Declare a business component interface for the session bean. This step is optional for EJB 3.1 local beans.
- Create the session bean class that implements the business interface if one exists.
- Configure the session bean by either annotating the session bean class or providing a deployment descriptor.

The following subsections describe these tasks in more detail.

Declaring a Business Interface for the Session Bean

An EJB 3.0 session bean component is required to use an interface to specify the business services it provides. Code 8-1 contains an example of a business service interface.

Code 8-1 Session Bean Business Interface

```
1   import javax.ejb.*;
2
3   @Remote
4   public interface HelloSession {
5     public java.lang.String hello();
6   }
```

Code 8-1 shows a plain Java technology interface that can be used as a business interface to create a session bean.

Creating the Session Bean Class that Implements the Business Interface

The EJB 3.0 specification requires a session bean class to implement the methods declared in its associated business interface. The bean class is also known as an implementation instance or EJB class. Code 8-2 shows a stateless session bean class implementing the remote business interface shown in Code 8-1

Code 8-2 A Stateless Session Bean Class

```
1   import javax.ejb.*;
2
3   @Stateless
4   public class HelloSessionBean implements HelloSession {
5
6      public java.lang.String hello() { return "Hello World!";}
7   }
```

Declaring Local and Remote Session Beans

The following frequently asked questions (FAQ) provide additional guidelines to assist you with the task of declaring session beans.

- What are the requirements for the session bean business interface?

  A session bean's business interface is a plain Java technology interface. The business interface can have super interfaces. The business methods can throw arbitrary application exceptions.

- How are business interfaces classified?

  A business interface to a session bean can be:

  - A local interface

    A local interface provides access to code that executes on the same Java Virtual Machine (JVM™) as the session bean component.

  - A remote interface

    A remote interface provides access to code that executes on a JVM that is different (remote) to that of the session bean component.

  A session bean can define both a local and a remote business interface.

- How can a business interface be designated a local interface?

  To specify an interface as a local interface, use the `Local` annotation on the interface or the bean class. For example:

```
@Local public interface Calculator {...}
@Stateless public class CalculatorBean implements Calculator {...}
```

Alternatively, you can use the `Local` annotation on the bean class. For example:

```
public interface Calculator {...}
@Local @Stateless public class CalculatorBean implements Calculator {...}
```

Or you use the default locality, which is local,

```
public interface Calculator {...}
@Stateless public class CalculatorBean implements Calculator {...}
```

- How can a business interface be designated a remote interface?

  To specify an interface as a remote interface, use the `Remote` annotation on the interface or the bean class. For example:

```
@Remote public interface Calculator {...}
@Stateless public class CalculatorBean implements Calculator {..}
```

  Alternatively, you can use the `Remote` annotation on the bean class. For example:

```
public interface Calculator {...}
@Remote @Stateless public class CalculatorBean implements Calculator {..}
```

- What are the requirements for a no-interface local view session bean?
  - The bean class has the same requirements as in the examples of a local interface bean
  - All EJB related annotations must be placed on the bean class as no interface exists

```
@Stateless public class CalculatorBean {...}
```

- Are there any additional requirements for remote interfaces?
  - The remote business interface is not required or expected to be a `java.rmi.Remote` interface.
  - A remote interface method must not expose local interface types, timers, or timer handles.
  - The arguments and return types of the methods must be of valid types for RMI/IIOP.
  - The throws clauses of the methods of the interface should not include the `java.rmi.RemoteException`.
- Can a session bean have more than one business interface?

  A session bean class can implement multiple interfaces. To designate an interface as the business interface of a bean class, use any of the following options:

- Local or remote annotation on the interface
- Local or remote annotation on the implementing bean class
- Local or remote designation in the deployment descriptor associated with the bean class

Requirements for a Session Bean Class

The session bean class must comply with the following design rules:

- The class must be a top-level class. In addition, the class must be defined as public, must not be final, and must not be abstract.
- The class must have a public constructor that takes no parameters. The container uses this constructor to create instances of the session bean class.
- The class must not define the `finalize` method.

The class must implement the methods of the beans business interface(s), if any.

Note – EJB 3.1 stateful session beans do not have a method to pass in initialization parameters during lookup and creation. You must pass any initialization data as parameters to a method call after lookup.

Annotating the Session Bean Class

The EJB 3.1 specification provides a number of annotations you can use to supply metadata about the session bean class. Code 8-3 shows an example of a stateless session bean class that demonstrates the use of annotations. You can identify the annotations by the leading `@` character that prepends each annotation.

Code 8-3 Session Bean Code With Annotation

```
1   @Stateless public class CalculatorBean implements Calculator {
2     public int add(int a, int b) {
3       return a + b;
4     }
5
6     public int subtract(int a, int b) {
7       return a - b;
8     }
9   }
```

Table 8-1 contains a brief explanation of the most commonly used annotations applicable to a session bean.

Table 8-1 Common Annotations Applicable to Session Beans

Annotation	Applies to	Description

`Stateful`
`Stateless`
`Singleton`
	Session bean class	The `Stateful, Stateless,` or `Singleton` annotation is used to denote the type of session bean. You are required to supply the session bean type by either annotating the bean class or by supplying the corresponding deployment descriptor element.

`Local` `Remote`	Session bean class or its business interface	The `Local` annotation designates a local interface of the bean. The `Remote` annotation designates a remote interface of the bean. The absence of `Local` or `Remote` implies that the bean class implements a local interface.
`Remove`	Stateful session bean class	The `Remove` annotation is used to denote a remove method of a stateful session bean.

The following types of annotations are also applicable to session beans.

- Transaction-related annotations

  Transaction-related annotations are described in ‘‘Implementing a Container-Managed Transaction Policy’’ on page 10-13”.

- Security-related annotations

  Security-related annotations are described in Module 15, “Implementing a Security Policy”.

- Life-cycle callback handler method annotation

  Life-cycle callback handler method annotations are described in ‘‘Life-Cycle Event Handlers’’ on page 8-12.

- Resource-related annotations

  While not specific to session beans the `@EJB` and `@Resource` annotations are often used in session beans.

Life-Cycle Event Handlers

With EJB 3.1, you are not required to provide life-cycle event handlers for session beans. However, session beans do generate life-cycle events and you can optionally provide event handlers for these methods.

A stateless session bean generates the following life-cycle events:

- `PostConstruct`
- `PreDestroy`

The stateless session bean life-cycle diagram shown in Figure 8-3 provides the context for the stateless session beans lifecycle events.

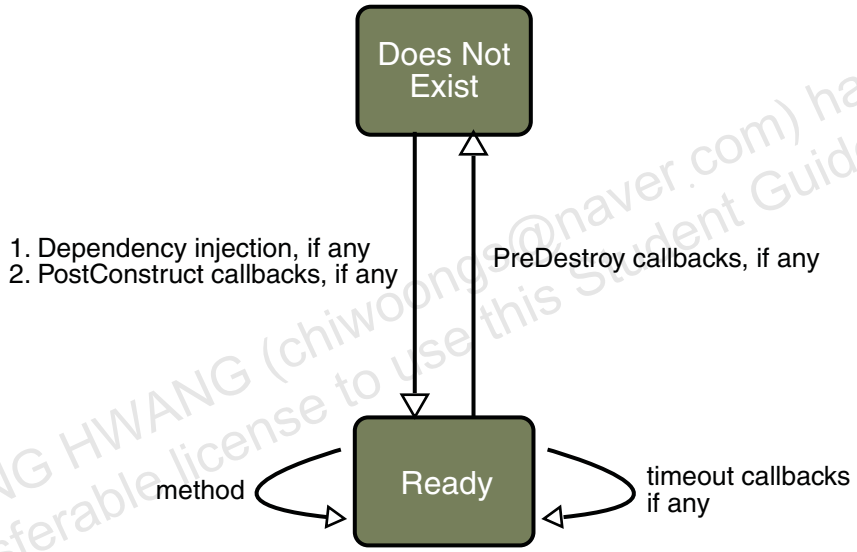


Figure 8-3 Life Cycle of a Stateless Session Bean

A stateful session bean generates the following life-cycle events:

- `PostConstruct`
- `PreDestroy`
- `PostActivate`
- `PrePassivate`

The stateful session bean life-cycle diagram shown in Figure 8-4 provides the context for the stateful session bean's life-cycle events.

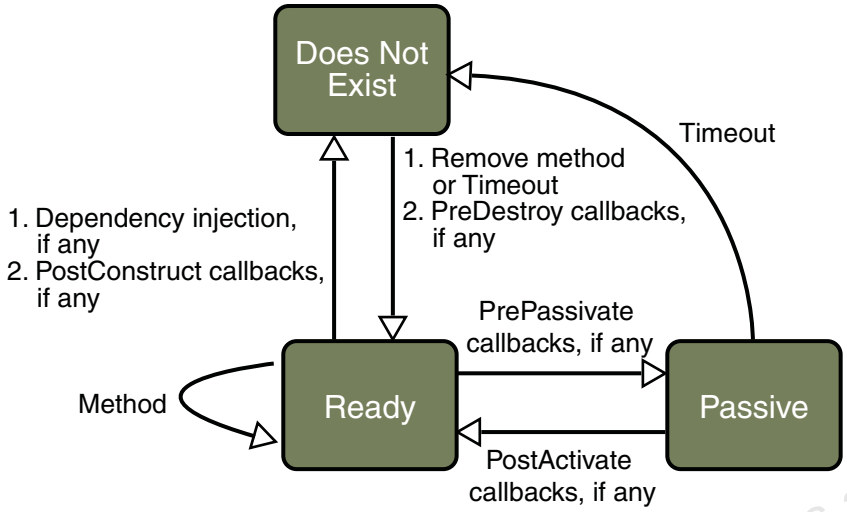


Figure 8-4 Life Cycle of a Stateful Session Bean

Code 8-4 shows an example of a life-cycle event handler method (`endShoppingCart`) defined within a bean class.

Code 8-4 Example of Callback Method in Bean Class

```
1   import javax.ejb.*;
2   import java.util.*;
3
4   @Stateful public class ShoppingCartBean implements ShoppingCart {
5     private float total;
6     private List productCodes;
7     public int someShoppingMethod(){...};
8     //...
9     @PreDestroy private void endShoppingCart() {...};
10  }
```

A callback method defined in a bean class is required to have the following signature:

```
anyAccessModifier void methodName()
```

Note – The method can have any name and any access modifier. The method's return type must be void and it must have zero arguments.

The SessionContext Object

Use the SessionContext object to access EJB objects, obtain current transaction status, and obtain security information. This information is used in later modules. SessionContext extends EJBContext.

```
1   import javax.ejb.*;
2   import javax.annotation.*;
3
4   @Stateful
5   public class BankingBean implements Bank {
6
7      @Resource private javax.ejb.SessionContext context;
8
9   }
```

Session Bean Packaging and Deployment

To package and deploy a session bean, complete the following tasks.

- Optionally, create a deployment descriptor file for the session bean component
- Package the bean in either a EJB component archive or Web component archive
- Deploy the archive containing the session bean component

Introducing Deployment Descriptors

This section provides an introduction to deployment descriptors.

- A deployment descriptor is an XML file that provides configuration information to the application server to configure the enterprise bean component during deployment.
- There are two main types of deployment descriptors:
  - Enterprise bean component deployment descriptors

    An enterprise bean component deployment descriptors file is named `ejb-jar.xml`. The `ejb-jar.xml` file contains deployment information for session, entity (EJB 2.1), and message-driven beans. EJB Component deployment descriptors contain the following types of configuration information.
    - Structural information

      Structural information describes the structure of an enterprise bean and declares an enterprise bean's external dependencies.
    - Application assembly information

      Application assembly information describes how the enterprise bean (or beans) in the `ejb-jar.xml` file is composed into a larger application deployment unit.

    A single `ejb-jar.xml` file can contain deployment information for one or more enterprise beans. Refer to chapter 19 of JSR 318: Enterprise JavaBeans™,Version 3.1 EJB for the XML schema for deployment descriptors associated with enterprise bean components.

- Java EE application deployment descriptors

  The Java EE deployment descriptors file is named `application.xml`. It contains deployment information for the Java EE application. Refer to The Java™ Platform, Enterprise Edition Specification, v6.0, Chapter 8 for the XML schema for deployment descriptors associated with Java EE applications.

- As of EJB 3.0, you can embed configuration information for enterprise components using metadata annotations in the bean component's source code.

  If all the configuration information needed for an enterprise bean is supplied using metadata annotations, you are no longer required to supply a deployment descriptor. You are required to supply a deployment descriptor if any of the required configuration information was not supplied using metadata annotations. Optionally, you can use a deployment descriptor to override any application assembly information originally embedded in the component's source code using metadata annotations. However, you must not override structural information.

- The deployment descriptor for an enterprise bean is part of the component archive that contains the bean. In addition to the deployment descriptor, an archive also contains all the enterprise bean classes.

Example of a Deployment Descriptor for an EJB

Code 8-5 contains an example of a deployment descriptor for a session bean.

Code 8-5 Session Bean Deployment Descriptor Example

```
1   <ejb-jar>
2     <enterprise-beans>
3       <session>
4         <ejb-name>ReportBean</ejb-name>
5         <business-remote>services.Report</business-remote>
6         <ejb-class>services.ReportBean</ejb-class>
7         <session-type>Stateful</session-type>
8         <transaction-type>Container</transaction-type>
9       </session>
10      <!-- Deployment information for additional beans goes here-->
11    </enterprise-beans>
12
13  <assembly-descriptor>
14    <!-- Add assembly information here-->
15  </assembly-descriptor>
16  </ejb-jar>
```

The deployment descriptor shown in Code 8-5 contains deployment information for a single session bean. You can add deployment information for additional enterprise beans at the same level as the `session` element.

EJB Packaging

Session bean components must be packaged in a deployable enterprise module to be deployed. Java EE 5 and previous applications require EJB components to be packaged in a EJB JAR file. The rules for EJB JAR files are outlined in the section, “Creating a EJB Bean Component Archive” on page 8-18.

EJB 3.1 beans can be packaged in a WAR file which eliminates the need for both EJB JAR and EAR files. When EJB components are packaged in a WAR file the deployment descriptor (`ejb-jar.xml`) is placed in the `WEB-INF` directory.

Creating a EJB Bean Component Archive

A EJB component archive is an EJB module, which is sometimes referred to as an EJB JAR file. You can create EJB JAR files using a GUI associated with an IDE or a Java EE application server. Alternatively, you can create the EJB JAR file using a command-line tool (utility), such as the Java technology archive tool `jar`. The following steps outline the process of creating an EJB JAR file using the `jar` utility.

1. Create a working directory structure.
   a. Create a root directory.
   b. Off the root directory, create the package directory structure corresponding to the EJB component classes and helper classes.
2. Copy the EJB component class files and helper classes into their corresponding subdirectories. For example, you should include the following classes:
   - The enterprise bean implementation class, business interfaces, and their super classes
   - All classes referenced by the enterprise bean's class, interfaces, and their super classes

   An EJB module's JAR file must *not* contain the following:
   - Java EE platform API classes and Java SE platform API classes

     For example, the JAR file must not contain classes, such as: `java.lang.String` or `javax.ejb.EJBException`.
3. Create a `META-INF` subdirectory off the root directory.
4. If a deployment descriptor is used to configure the enterprise bean, then copy the deployment descriptor file (`ejb-jar.xml`) for the EJB components into the `META-INF` directory.

   You need to examine the deployment descriptor file to ensure that all required elements have been supplied for the EJB components included in the EJB JAR archive.

   Most application servers also require you to include a vendor-specific deployment descriptor in the `META-INF` directory. For example, the GlassFish application server names this file `sun-ejb-jar.xml`.
5. From a command line, execute the `jar` utility to create the EJB JAR archive.

   If you examine the created EJB JAR archive you see that it includes a manifest file called `MANIFEST.MF` in the `META-INF` directory.

Figure 8-5 illustrates the directory structure and class files used to create the auction system EJB module's JAR file.

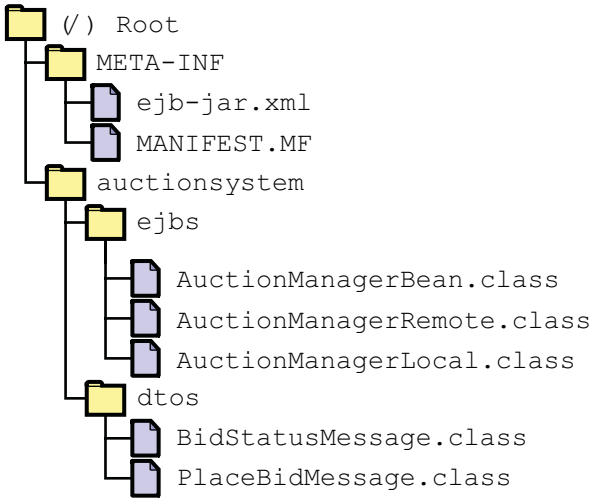


Figure 8-5 Example of the EJB Module JAR File

Deploying a Session Bean Component Archive

Deploying a session bean component archive, or for that matter any EJB module, is specific to the application server. Most application servers provide the following two deployment options.

- IDE-based deployment support
- Command-line based deployment support

For additional information on deployment, refer to the documentation associated with the application server you are using.

Creating a Session Bean Client

Before proceeding to the task of creating session bean clients, you should examine the answers to the following questions.

- What are some examples of session bean clients?
  - Another enterprise bean, such as another session or entity bean.
  - A web component, such as a servlet.
  - A plain Java class, such as an application client.
- What factors should be considered when writing a session bean client?

  The most important factor is the availability of standard container services, in particular naming lookup and injection services. These services are supplied as standard by the application server container for Java EE components.

  In the case of a plain Java class, such as an application client, you can use an application client container to provide these services. Alternatively, you can obtain the services an application client requires from Java SE class methods, such as methods from the `javax.naming.InitialContext`.
- What is an application client container?

  An application client container is a set of classes that work together to provide a hosting environment for an application client. The hosting environment includes a JVM and a set of services. These services include security, naming, injection, and communication (with the application server container) services. A client using the application container executes in a different JVM to that of the application server.

  The packaging of the classes that make up the application client container is vendor specific. Some vendors provide an application client container as a JAR file. Application client containers are installed on the client machine.

The following sections examine the tasks of creating a session bean client in the following contexts:

- Creating a client using container services
- Creating a client without using container services

Creating a Client Using Container Services

The following steps outline a process for creating a client that uses container services to access the services of a session bean. Any managed component can use container services. Managed components include Session EJBs, Message EJBs, and Servlets.

1. Use injection services of the container to obtain a reference to the session bean object.
2. Invoke the required services of the session bean object.

Code 8-6 contains an example of a client that uses container services to access a session bean.

Code 8-6 Container-Based Session Bean Client

```
1   import javax.ejb.*;
2
3   public class InternalClient {
4     @EJB                                  // step 1
5     private static Calculator calc;
6
7     public static void main (String args[]) {
8       int sum = calc.add(1, 2);           // step 2
9     }
10  }
```

Creating a Client Without Using Container Services

The following steps outline a process for creating a session bean client that is external to a container.

1. Use JNDI naming services to obtain a reference to the session bean object.
2. Invoke the required services of the session bean object.

Code 8-7 contains an example of a client that uses JNDI naming services to access a session bean.

Code 8-7 Non-Container-Based Session Bean Client

```
1   import javax.naming.*;
2   import javax.rmi.*;
3
4   public class ExternalClient {
5     public static void main (String args[]) {
6       // step 1 begin
7       Context ctx = new InitialContext();
8       Object obj = ctx.lookup("ejb/Calculator");
9       Calculator calc = (Calculator)
10        PortableRemoteObject.narrow(obj, Calculator.class);
11      // step 1 end
12      int sum = calc.add(1, 2);       // step 2
13    }
14  }
```

Session Bean Client Reference Types

- If a session bean has a business interface(s) the client will use the interface as the reference type to the session bean.
- If a session bean has no business interface (no-interface view) then the client will use the bean class as the reference type to the session bean.
- Clients do not instantiate the bean directly, regardless of reference type.

Portable JNDI Session Bean Clients

The 3.0 and previous EJB specifications do not require session beans to have a standard JNDI name, although many application servers do provide one. To perform portable JNDI lookups of a session bean requires editing the `web.xml` file or the `ejb-jar.xml` file depending on the client type to create an indirect JNDI reference. An example of a web-tier component would be a plain Java helper class, such as a business delegate.

```
<ejb-ref>
    <ejb-ref-name>BrokerLookup</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <remote>trader.BrokerModel</remote>
</ejb-ref>
```

```
try {
   Context ctx = new InitialContext();
   model = (BrokerModel)ctx.lookup("java:comp/env/BrokerLookup");
} catch(NamingException ne) {...}
```

Global JNDI Names

The Environmental Naming Context (ENC) is a JNDI context that can be accessed via `java:comp/env`. This context is used to create portable JNDI names for EJBs. Names in the ENC are created with entries in a deployment descriptor.

The Java EE 6 specification adds support for standard JNDI names of EJB components without requiring the use of the ENC.

```
java:global[/<app-name>]/<module-name>/<bean-name>
java:app[/<module-name>]/<bean-name>
java:module/<bean-name>
```

The `app-name` is the same as the EAR name, minus extension. The `module-name` is the EJB jar or WAR file name, minus extension. The `java:app` context only looks up components within the same application as the client. The `java:module` context only looks up components within the same module as the client.

Dependency injection annotations can target a specific JNDI name.

```
@EJB(mappedName="java:global/fooejb/FooBean")
FooRemote foo;
```

Summary

This module describes how you can implement EJB 3.1 session beans. The key issues can be summarized as:

1. Session bean types: stateful, stateless, and singleton
2. Declare the business interface for the session bean.
3. Create the session bean class that implements the business interface.
4. Annotate the session bean class.
   a. Session bean type
   b. Local or remote
   c. Dependency injection
   d. Transaction
   e. Security
5. If required, add handling code for lifecycle callback events.
6. Package the session bean.
7. Deploy the session bean.
8. Create the client.

Module 9

The Java Persistence API

Objectives

Upon completion of this module, you should be able to:

- Describe the role of the Java Persistence API (JPA) in a Java EE application
- Describe the basics of Object Relational Mapping
- Describe the elements and environment of an entity component
- Describe the life cycle and operational characteristics of entity components

Additional Resources

Additional resources – The following references provide additional information on the topics described in this module:

- Eric Jendrock, Debbie Carson, Ian Evans, Devika Gollapudi, Kim Haase, Chinmayee Srivathsa. “The Java EE 6 Tutorial,” [`http://java.sun.com/javaee/6/docs/tutorial/doc/`], accessed 1 August 2009.
- "JSR 317: Java Persistence, Version 2.0, Java Persistence API", [`http://jcp.org/en/jsr/detail?id=317`], accessed 1 August 2009.

The Java Persistence API

This section uses a FAQ list to examine the features of Java persistence, as defined in the Java persistence specification (JSR 317).

- What is data persistence?

  Data persistence is the mechanism used by applications to preserve application data (that would be lost by application shutdown or computer power down) in a persistence store, such as a database.

- What is the Java persistence specification?

  The Java persistence specification is the specification of the Java API for the management of persistence and object/relational mapping with Java EE and Java SE. The technical objective of the specification work is to provide an object/relational mapping facility for the Java application developer using a Java domain model to manage a relational database.

- How does the Java Persistence API relate to the Java EE application servers?

  All Java EE application servers are required to provide an implementation of the Java Persistence API but the Java Persistence API may be used without a Java EE application server. The two types of persistence are:

  - Container-Managed Persistence
  - Application-Managed Persistence

Object Relational Mapping Software

The de facto persistence mechanism used in most enterprise applications is a relational database. Object-oriented program design and relational database table structure might not organize data in the exact same structure. A Java business domain object can encompass partial data from a single database table or include data from multiple tables depending on the normalization of the relational database.

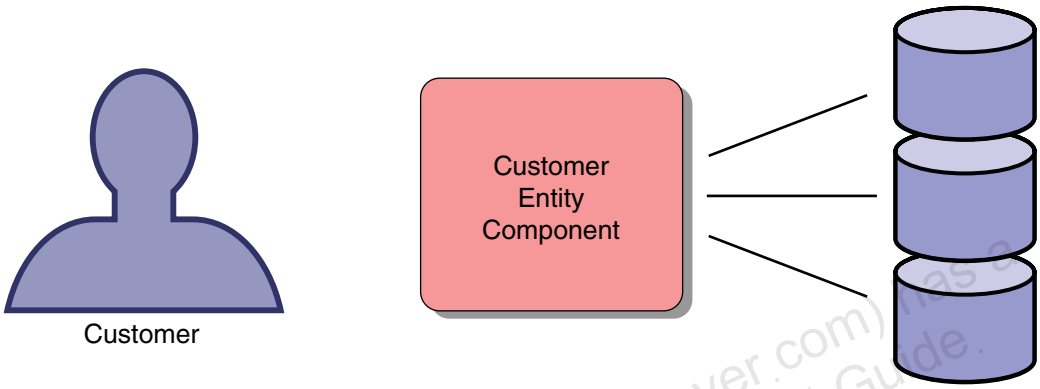


Figure 9-1 Business Concept-to-Object Oriented Domain Object and Relational Database Mapping

Writing code to translate from an object-oriented domain scheme to a relational database scheme can be a time consuming endeavor. Object Relational Mapping (ORM) software attempts to provide this mapping to OO software developers without requiring much or any coding. Often just configuration information in the form of code annotations or XML configuration files are supplied to ORM software. Examples of existing ORM software are EclipseLink and Hibernate.

Note – EclipseLink, `http://www.eclipse.org/eclipselink/`, is the continuation of Oracle's open source version of Toplink. Oracle donated the source code for Toplink, a JPA 1.0 provider, to the Eclipse Foundation. EclipseLink is the reference implementation for the JPA 2.0 specification.

The most basic ORM software supports a simple mapping of Java objects to database tables, as shown in Figure 9-2.

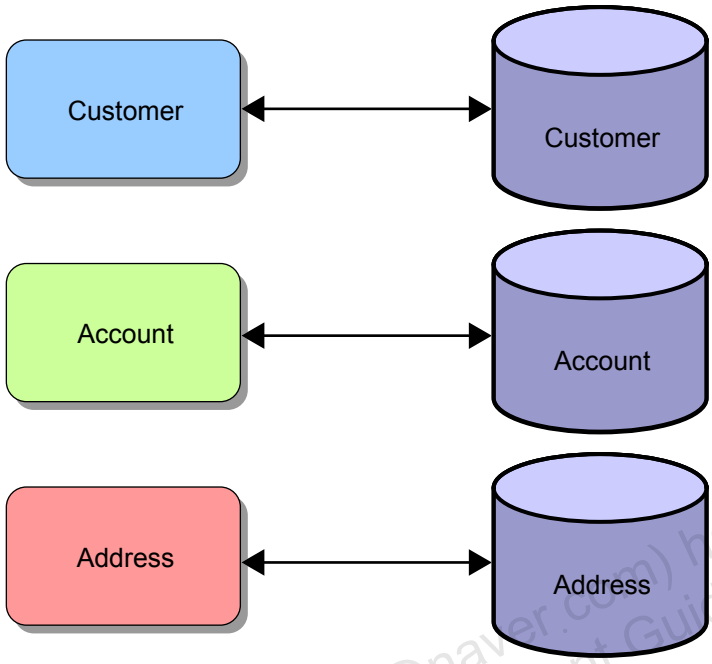


Figure 9-2 One-to-One Mapping of Java Objects to Relational Tables

Modern ORM software has the ability to map Java objects to database table structures that do not have a simple one-to-one mapping as shown in Figure 9-3.

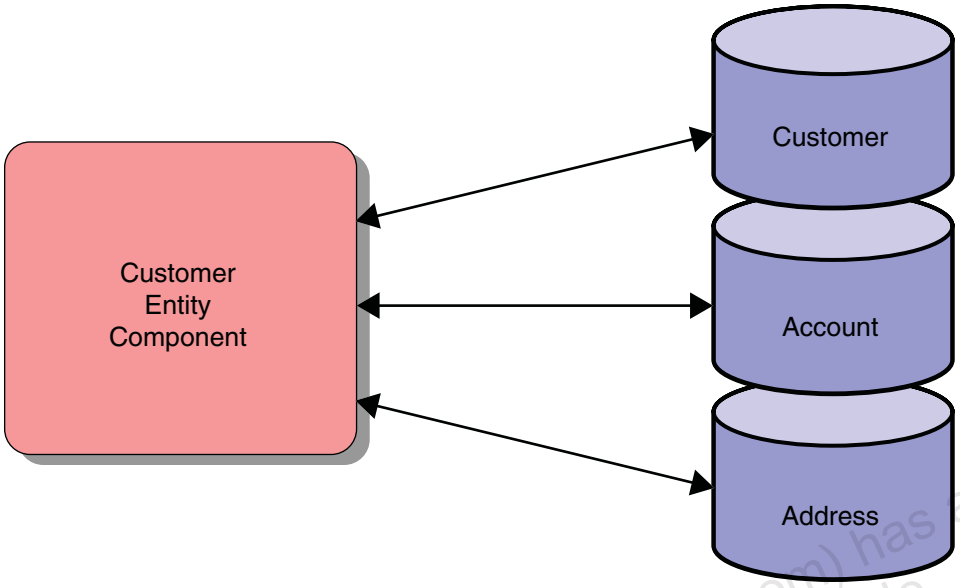


Figure 9-3 One-to-Many Mapping of a Java Object to Relational Tables

J2EE 1.4 application servers provide EJB 2.1 Container Managed Persistence (CMP) EJBs as the standard ORM technique. Other ORM methods, such as Java Data Objects (JDO) also exist. The Java Persistence API attempts to combine the best features from existing ORM technologies while removing the application server requirement of EJB 2.1 CMP EJBs.

The Java Persistence API specification is just a specification for an API and life cycle behavior, it is not usable ORM software. A Java Persistence API provider is required to use the Java Persistence API. Two examples of a Java Persistence API provider are EclipseLink and Hibernate.

Entity Class Requirements

This section provides an overview of the tasks required to define a Java Persistence API entity class. To define an entity class, you are required to perform the following tasks:

- Declare the entity class.
- Verify and override the default mapping.

The following subsections describe these tasks in more detail.

Declaring the Entity Class

The following steps outline a process you can use to declare an entity class.

1. Collect information required to declare the entity class

   This step involves identifying the application domain object (identified in the object oriented analysis and design phase of application development) that you want to persist.

   - Use the domain object name as the class name.
   - Use the domain object field names and data types as the field names and types of entity classes.

2. Declare a public Java technology class.

   The class must not be final, and no methods of the entity class can be final. The class can be concrete or abstract. The class can not be an inner class.

3. If an entity instance is to be passed by value as a detached object through a remote interface, then ensure the entity class implements the `Serializable` interface.
4. Annotate the class with the `javax.persistence.Entity` annotation.
5. Declare the attributes of the entity class.

   Attributes must not have public visibility. They can have private, protected, or package visibility. Clients must not attempt to access an entity classes' attributes directly.

6. You can optionally declare a set of public getter and setter methods for every attribute declared in the previous step.
7. Annotate the primary key field or the getter method that corresponds to the primary key column with the `Id` annotation.

8. Declare a public or protected no-arg constructor that takes no parameters. The container uses this constructor to create instances of the entity class. The class can have additional constructors.

Code 9-1 shows an example of an entity class.

Code 9-1 Entity Class Code Example

```
1   import java.io.Serializable;
2   import javax.persistence.*;
3
4   @Entity @Table(name = "TABLE1")
5   public class MyEntity implements Serializable {
6
7      @Id @Column(name = "ID") private int id;
8      @Column(name = "MSG") private String message;
9
10     protected MyEntity() { }
11     public MyEntity(int id, String message) {
12        this.id = id;
13        this.message = message;
14     }
15
16     public int getId() { return id; }
17     public String getMessage() { return message; }
18     public void setMessage(String message) { this.message = message; }
19  }
```

Verifying and Overriding the Default Mapping

The following table outlines the default standard relationship between entity classes and relational databases.

Table 9-1 Mapping Object Tier Elements to Database Tier Elements

Object Tier Element	Database Tier Element
Entity class	Database table
Field of entity class	Database table column
Entity instance	Database table record

The default database table name can be overridden using the `Table` annotation. The Java Persistence API assigns the name of the entity class as the default name to the database table. For example:

Code 9-2 Table Mapping

```
@Entity
@Table(name = "Cust")   //Cust is the name of the database table
public class Client {
  //...
}
```

The Java Persistence API assigns the name of the entity class property (or field) name as the default name to the database table column. Use the `Column` annotation to override the default setting. For example:

Code 9-3 Column Mapping

```
@Entity
@Table(name = "Cust")
public class Client {
  @Column(name = "cname")
  private String clientName;
  //...
}
```

You have several options available for generating the values of primary keys. The simplest of these is to use the auto generation feature of the container, as shown in the following example.

Code 9-4 Primary Key Generation

```
@Entity
@Table(name = "Cust")
public class Client {
  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private int clientReference;
  //...
}
```

When using `@GeneratedValue(strategy = GenerationType.AUTO)` Toplink creates a sequence table in the relational database. Other JPA providers can use different techniques to create unique IDs.

Note – This discussion examined the optional task of overriding the default settings. For the vast majority of situations, using the defaults should be sufficient if no previous database tables exist.

Persistent Fields as Opposed to Persistent Properties

Entity classes have their state synchronized with a database. The state of an entity class is obtained from either its variables (fields) or its accessor methods (properties). Field or property-based access is determined by the placement of annotations. A single persistence annotation on one field (variable) means that the entire persistence state is determined by the entity object's variables. JPA 1.0 entities cannot have both field and property based access.

Note – JPA 2.0 allows the mixing of field based and property based access with the introduction of the `javax.persistence.Access` annotation.

Persistent Fields

When using persistent fields, the persistence provider retrieves an object's state by reading its variables.

- Persistent fields cannot be public
- Persistent fields should not be read by clients directly
- Unless annotated with `@Transient` or modified with the transient keyword, all fields are persisted regardless of whether they have a `@Column` annotation.

Code 9-5 Persistent Fields

```
@Id @Column(name = "ID") private int id;
@Column(name = "MSG") private String message;

public int getId() { return id; }
public String getMessage() { return message; }
public void setMessage(String message) { this.message = message; }
```

Persistent Properties

When using persistent properties, the persistence provider retrieves an object's state by calling its accessor methods.

- Methods must be public or protected
- Methods follow the JavaBeans naming convention
- Persistence annotations can only be on getter methods

Code 9-6 Persistent Properties

```
private int id;
private String message;

@Id @Column(name = "ID") public int getId() { return id; }
public void setId(int id) { this.id = id; }

@Column(name = "MSG")
public String getMessage() { return message; }
public void setMessage(String message) { this.message = message; }
```

Persistence Data Types

Regardless of the use of field or property-based access, the same rules apply governing the Java data types that can be mapped to SQL data types. Data types that can be mapped include:

- Java primitive types
- Java wrappers, such as `java.lang.Integer`
- `java.lang.String`
- `byte[]` and `Byte[]`
- `char[]` and `Character[]`
- Any serializable types including but not limited to:
  - `java.util.Date`
  - `java.sql.TimeStamp`

Note – Complex data types that do not have simple mapping, such as a `java.lang.String` to a VARCHAR or CHAR column might require additional annotations. These annotations are described in SL-370: *Building Database-Driven Applications With Java Persistence API* .

The Concept of a Primary Key

An entity component distinguishes itself and the data it represents from other entities using a unique ID value known as a primary key. Primary keys:

- Give an entity instance its persistent identity. An entity identity is a concept used by the entity manager, which is discussed later.
- Are typically a string or an integer but can also be custom classes that correspond to several database table columns. The most common table design is to have an INT column with a UNIQUE constraint.
- Are in every Entity class. On the database side, it might be required that the primary key columns are INDEX columns.
- Can be auto incrementing.

```
@Id @GeneratedValue(strategy = GenerationType.IDENTITY)
@Column(name = "ID")
private int id;
```

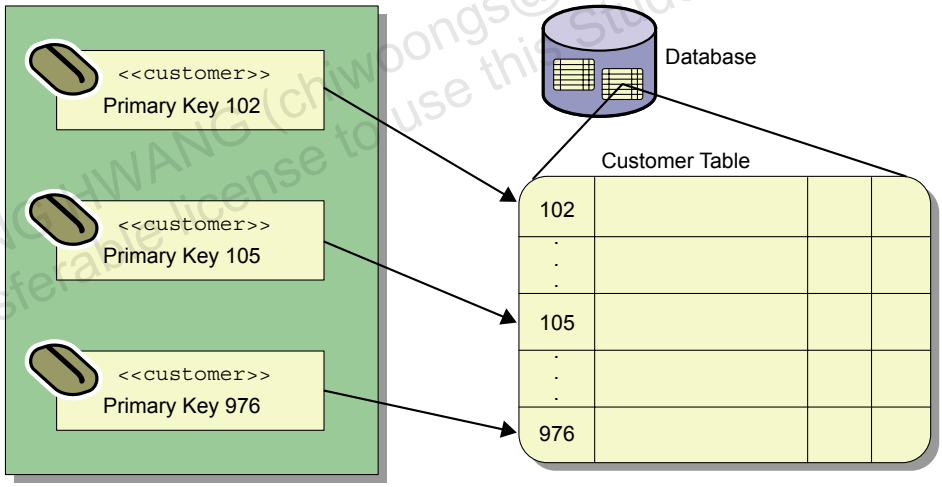


Figure 9-4 Relationship Between Entity Instances and Tables Rows

Life Cycle and Operational Characteristics of Entity Components

Besides entity classes, there are several key concepts that must be understood to begin leveraging the Java Persistence API. They are:

- Persistence Unit

  Configuration information in the form of an XML file and a bundle of classes that are controlled by the Java Persistence API provider.

- Entity manager

  An entity manager is the service object that manages the entity life-cycle instances. This is the core object that a Java developer uses to create, retrieve, update, and delete data from a database.

- Persistence context

  Every entity manager is associated with a persistence context. A persistence context is a set of entity instances in which, for any persistent entity identity, there is a unique entity instance.

- Persistent identity

  The persistent identity is a unique value used by the container to map the entity instance to the corresponding record in the database. Persistent identity should not be confused with Java object identity.

Persistence Units

A Persistence unit is a collection of entity classes stored in a EJB-JAR, WAR, or JAR archive along with a `persistence.xml` file. ''Deploying Entity Classes'' on page 9-21 describes the rules and placement of entity classes in detail. The key concept to understand at this point is that a persistence unit defines what entity classes are controlled by an entity manager. A persistence unit is limited to a single DataSource.

The `persistence.xml` file

Configuration of a persistent unit is controlled by a XML configuration file named `persistence.xml`. The `persistence.xml` file:

- Configures which classes make up a persistence unit
- Defines the base of a persistence unit by its location
- Specifies the DataSource used

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
xmlns="http://java.sun.com/xml/ns/persistence">
  <persistence-unit name="BrokerTool-ejb" transaction-type="JTA">
    <jta-data-source>StockMarket</jta-data-source>
    <jar-file>BrokerLibrary.jar</jar-file>
    <properties/>
  </persistence-unit>
</persistence>
```


The Persistence Context

A persistence context can be thought of as a working copy of a persistence unit. Several persistence contexts using the same persistence entity can be active at the same time. A persistence context:

- Typically lasts the duration of a transaction
- Limits entity instances to a single instance per persistent identity
- Has a management API known as the entity manager

The Entity Manager

An entity manager provides methods to control events of a persistence context and the life cycle of entity instances in a persistence context. An entity manager:

- Provides operations, such as `flush()`, `find()`, and `createQuery()`, to control a persistence context
- Replaces some of the functionality of home interfaces in EJB 2.1 entity beans

An entity manager can be obtained using dependency injection in managed classes.

```
@PersistenceContext private EntityManager em;
```

Entity Instance Management

When a reference to an entity manager has been obtained, you use its methods to marshall entity instance data into and out of the database.

Note – Many exceptions that can be thrown by the `EntityManager` methods are indirect subclasses of `java.lang.RuntimeException`. IDEs and compilers do not require try catch blocks around lines of code that cause a `RuntimeException` subclass, such as `javax.persistence.EntityExistsException`, to be thrown.

Code 9-7 EntityManager Example

```
1   import javax.persistence.*;
2   import javax.ejb.*;
3   @Remote @Stateless
4   public class BrokerModelImpl implements BrokerModel {
5   @PersistenceContext private EntityManager em;
6   public Stock getStock(String symbol) throws BrokerException {
7           Stock stock = em.find(Stock.class, symbol);
8           if(stock != null) {
9              return stock;
10          } else {
11             throw new BrokerException("Stock : " + symbol +
12                " not found");
13          }
14  }
15
16  public void addStock(Stock stock) throws BrokerException {
17          try {
18              em.persist(stock);
19          }catch(EntityExistsException exe) {
20              throw new BrokerException("Duplicate Stock : " +
21                  stock.getSymbol());
22          }
23  }
24
25  public void updateStock(Stock stock) throws BrokerException {
26          Stock s = em.find(Stock.class, stock.getSymbol());
27          if(s == null) {
28               throw new BrokerException("Stock : " +
29                     stock.getSymbol() + " not found");
30          } else {
31               em.merge(stock);
32          }
33  }
34  public void deleteStock(Stock stock) throws BrokerException {
35          String id = stock.getSymbol();
36          stock = em.find(Stock.class, id);
37          if(stock == null) {
38              throw new BrokerException("Stock : " +
39             stock.getSymbol() +" not found");
40          } else {
41              em.remove(stock);
42          }
43  }
44  }
```

Entity States and Entity Manager Methods

JPA Entities have a transitional life cycle. An entity can exist in one of four states that are controlled primarily by the entity manager.

Figure 9-5 shows the possible states of an entity instance. The `persist`, `merge`, and `remove` methods are part of the entity manager API.

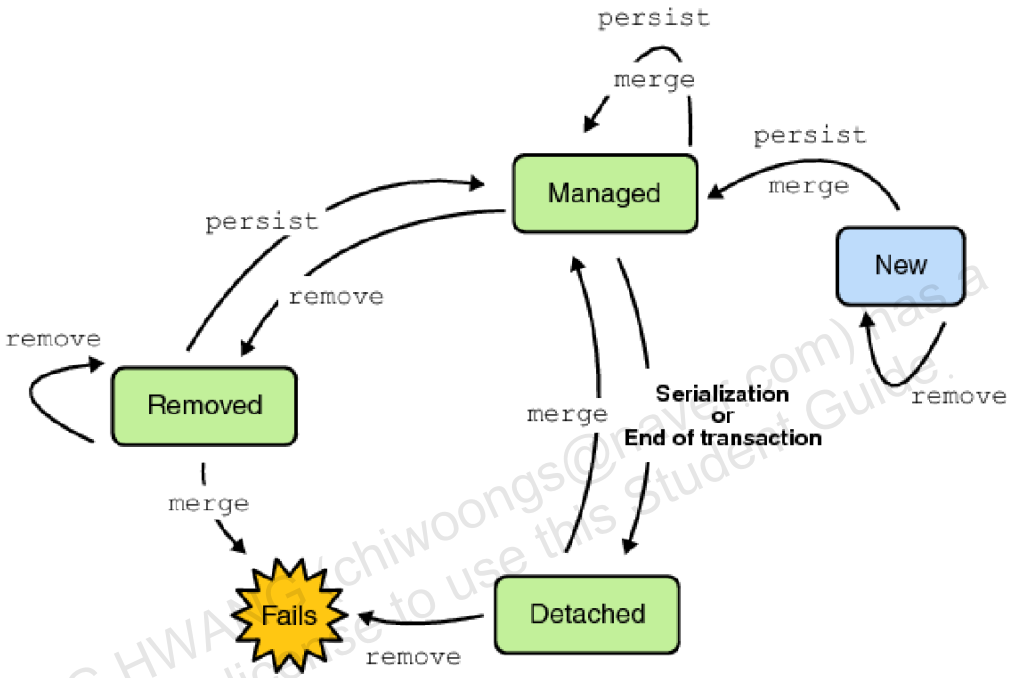


Figure 9-5 Entity Instance States and the Entity Manager Methods That Control Them.

Table 9-2 provides a summary of the significance of each entity state shown in Figure 9-5.

Table 9-2 Entity Life-Cycle States

Entity State	Significance
`New`	A new instance of the entity created using the `new` keyword. There is no corresponding database record in the persistence tier associated with the entity's primary key and the entity instance is not connected to a persistence context.
`Managed`	There is a corresponding database record and data in the record is kept synchronized (by the persistence provider) with data in the entity. The entity instance is connected to a persistence context and has a unique entity identity. Only one managed instance of an identity can exist in a persistence context.
`Detached`	There is a corresponding database record but data in the record and data in the entity are not synchronized and the entity instance is not connected to a persistence context.
`Removed`	This state represents a pending removal of the corresponding data record in the database.

Table 9-3 shows the entity manager methods or events required to effect the change of an entity life-cycle state.

Table 9-3 Changing Entity Life-Cycle States

Old Entity State	Required Operation	New Entity State
Does not exist	Use the `new` operator to create a new instance.	`New`
`New`	Use the entity manager's `persist` operation.	`Managed`
`Managed`	Use the entity manager's `remove` operation.	`Removed`
`Managed`	The consequence of the persistence context ending. Instances generated by serialization or cloning (of managed instances) are always detached.	`Detached`
`Removed`	Use the entity manager's `persist` operation.	`Managed`
`Removed`	Lose the reference to the removed instance.	Non existent

Entity Manager Methods

Table 9-4 shows several useful entity manager methods.

Table 9-4 Changing Entity Life-Cycle States

Entity Manager Method	Comment
`flush`	Forces the synchronization of the database with entities in the persistence context.
`refresh`	Refreshes the entity instances in the persistence context from the database.
`find`	Finds an entity instance by executing a query by primary key on the database.
`contains`	Returns true if the entity instance is in the persistence context. This signifies that the entity instance is managed.
`merge`	Merge the state of the given entity into the current persistence context.
`remove`	Remove the entity instance.
`persist`	Make an entity instance managed and persistent.

Managed Entities

A Managed entity will have any changes to its persistent fields or properties persisted. For example, the "`EntityManager` Example" on page 9-17 could be modified to change the values in the persistent properties of a managed object instead of using `merge`.

```
public void updateStock(Stock stock) throws BrokerException {
        Stock s = em.find(Stock.class, stock.getSymbol());
        if(s == null) {
               throw new BrokerException("Stock : " +
                     stock.getSymbol() + " not found");
        } else {
               //em.merge(stock);
               s.setPrice(stock.getPrice());
        }
}
```

Deploying Entity Classes

This section examines the issues and tasks associated with deploying entity classes. It uses a series of leading questions to present the information you require.

- What is required to deploy entity classes?

  To deploy entity classes, you are required to create a persistence unit.

- What is a persistence unit?

  A persistence unit is a logical grouping of all the elements required by the container to support the persistence management of an application.

  Table 9-5 shows the list of required components of a persistence unit.

Table 9-5 Components of the Persistent Unit

Persistence Unit Component	Comment
All managed (entity) classes. Object relational mapping metadata	These classes define what is persisted. This mapping defines the mapping between the managed classes and database tables.
Entity manager factory and entity managers	This information specifies which entity manager factory and which entity managers are to be used.
Configuration information for the entity manager factory and entity managers	The configuration information defines how the entity manager factory and entity managers are to be configured.
A `persistence.xml` file	This file defines the persistence unit.

- Is a persistence unit a standalone deployable module?

  The persistent unit is not a standalone deployable unit. Instead, the components that make up the persistent unit are placed in selected deployable Java EE modules.

- Where should the components of the persistence unit be placed?

  For deployment to a Java EE application server, the components of the persistence unit must be placed in one of the following locations:

  - In a EAR file
  - In a EJB-JAR file
  - In a WAR file
  - In an application client JAR file

- Within an EJB-JAR file, at which locations should the components of the persistence unit be placed?

Table 9-6 Default Locations for Persistence Unit Components

Persistence Unit Component	**Default Location**
All Managed (entity) classes. Object relational mapping metadata	In directories that map the package structure of the managed classes. The directory path root is the root of the EJB-JAR file. These are included as annotations in the entity classes. An alternative default location is the XML deployment descriptor file.
Entity manager factory and entity managers	The default entity manager factory and default entity managers are placed in a persistence provider specific location. These are known to the Java EE application server.
Configuration information for the entity manager factory and entity managers	The default configuration information is known to the Java EE application server.
The `persistence.xml` file	In the `META-INF` directory off the root of the EJB-JAR file.

Creating a Persistence Unit Using Default Settings

Figure 9-6 shows an example of persistence unit components in default settings within an EJB-JAR. It also shows other components of an EJB-JAR archive.

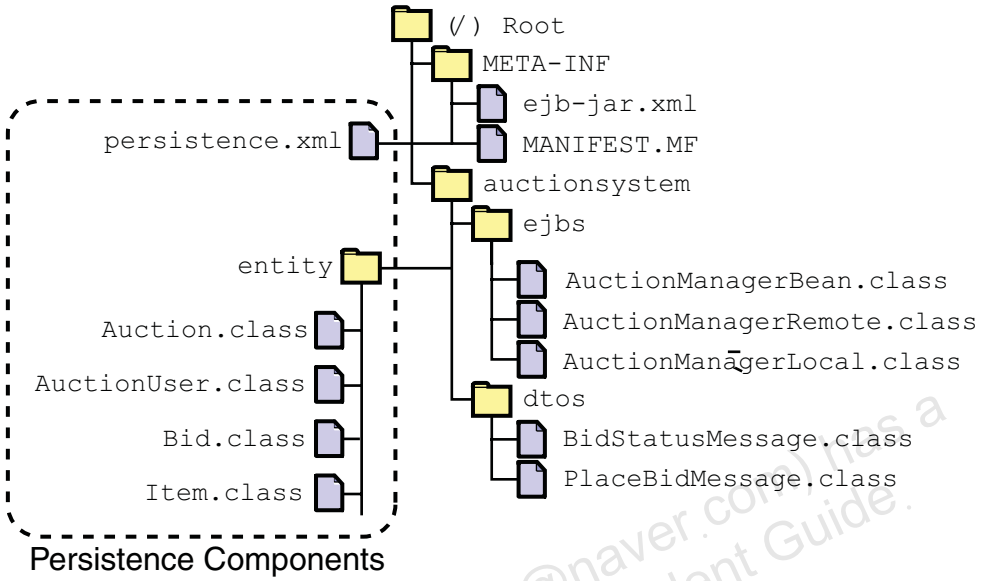


Figure 9-6 Example of Persistence Unit Components in Default Settings

The following steps outline a process you can follow to create a persistence unit within in an EJB-JAR file using the default settings for various components of the persistence unit. This process assumes that the entity classes are fully annotated and all classes are placed in the root directory.

1. If not already expanded, expand the EJB-JAR archive.
2. Create subdirectories that reflect the full qualified names of the managed classes you need to include in the persistence unit.
3. Copy all the classes you need into the directories created in Step 2.
4. Create a minimal `persistence.xml` file.

   At a minimum, you must provide a XML file with the following elements.

   - The `persistence` element
   - The `persistence-unit` element with a value for the `name` attribute.

   Code 9-8 shows the listing of the minimum content of the `persistence.xml` file.

Code 9-8 Example of the Minimal Persistence XML file

```
1   <persistence>
2     <persistence-unit name="OrderManagement"/>
3   <persistence>
```

5. Copy the `persistence.xml` file into the `META-INF` directory off the EJB-JAR root directory.
6. Create (archive) the EJB-JAR module.

Examining a Persistence Unit Using Non-Default Settings

Note – This shows the **optional** task of using non-default settings to create a persistence unit. For the vast majority of situations, using the defaults should be sufficient. JPA 2.0 provides new APIs to control cache behavior.

Code 9-9 The `persistence.xml` File With Non-Default Settings

```
1   <?xml version="1.0" encoding="UTF-8" ?>
2   <persistence version="2.0"
xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
3
4     <persistence-unit name="BrokerTool-ejb" transaction-type="JTA">
5       <jta-data-source>StockMarket</jta-data-source>
6       <jar-file>BrokerLibrary.jar</jar-file>
7       <mapping-file>mapping.xml</mapping-file>
8       <properties>
9         <property name="eclipselink.cache.type.default" value="NONE"/>
10      </properties>
11    </persistence-unit>
12  </persistence>
```

- The JAR files off root

  The JAR files off the root are optional. They are an alternative method for the developer to provide the managed set of entity classes. Every JAR file provided requires a corresponding `jar-file` element in the `persistence.xml` file. The presence of the `jar-file` elements in the `persistence.xml` file prompts the application server to load the classes archived in the JAR file.

- The mapping XML files off root

  The mapping XML files off the root are an alternative location for the object/relational mapping information. These mapping files require a corresponding `mapping-file` element in the `persistence.xml` file.

  The mapping XML files are optional. All mapping information can be supplied through the appropriate annotations in the entity class.

- `<properties>` are used to pass configuration information to the persistence provider.

Java SE JPA Applications

The examples so far in this module assume that JPA is being used by a session EJB. Using JPA in an EJB container has certain benefits:

- Transactions are taken care of automatically. Transactions are covered in more detail in "Implementing a Transaction Policy" on page 10-1.
- There is less configuration because the persistent unit is using a preconfigured `DataSource` for database access.
- Dependency injection is used to obtain an `EntityManager` reference.

With just a little extra work, JPA can be used in non-EJB applications such as Web applications or Java SE desktop applications. The following examples show the important changes to JPA configuration and code in a SE application:

Code 9-10 An example `persistence.xml` file for a Java SE JPA application.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">

<persistence-unit name="StockPU" transaction-type="RESOURCE_LOCAL">

     <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
        <class>trader.Stock</class>
        <class>trader.Customer</class>
        <class>trader.CustomerShare</class>
        <properties>
          <!-- useful log levels include OFF,FINE,SEVERE -->
          <!-- ddl-generation values are create-tables and drop-and-create-tables -->
          <property name="javax.persistence.jdbc.user" value="public"/>
          <property name="javax.persistence.jdbc.password" value="public"/>
          <property name="javax.persistence.jdbc.url"
value="jdbc:derby:MyDB;create=true"/>
          <property name="javax.persistence.jdbc.driver"
value="org.apache.derby.jdbc.EmbeddedDriver"/>
          <property name="eclipselink.ddl-generation" value="drop-and-create-tables"/>
          <property name="eclipselink.logging.level" value="SEVERE"/>
        </properties>
     </persistence-unit>
</persistence>
```

Note – The `persistence.xml` file example shown on page 26 uses the embedded Derby database driver which automatically starts a database instance within the same JVM as the application.

Code 9-11 Programmatically obtaining an `EntityManager`.

```
EntityManagerFactory emf =
Persistence.createEntityManagerFactory("StockPU");
EntityManager em = emf.createEntityManager();
EntityTransaction entityTransaction = entityManager.getTransaction();
entityTransaction.begin();
// ...
entityTransaction.commit();
```

In order to use JPA in a Java SE application several JAR files must be added to a project. In the `CLASSPATH` the application will require:

- The library JAR file(s) that contain the persistence provider. Both Hibernate and EclipseLink can be freely downloaded.
- The library JAR files to satisfy any required dependencies of the persistence provider. There can be multiple dependencies that must be meet. Hibernate requires several additional library JAR files.
- A running database and its JDBC driver. Derby, also known as the JavaDB, is a popular choice for small databases because of its portability and embedded option.

Advanced Persistence Features

Advanced persistence features covered in SL-370: *Java Persistence API 2.0*:

- Lifecycle Callbacks - Entity classes can contain callback methods that are executed by the JPA provider during lifecycle events.
- Relationships - The Persistence API can be used to manage foreign key tables and columns in a RDBMS. In Java these database relationships are modeled with references and collections.
- Java Persistence Query Language - A portable database query dialect used to avoid adding native SQL to Java applications.
- Queries - Come in two types, `NamedQueries` and `NativeQueries`. Queries are used to execute Query Language or native SQL statements on a database.
- The Criteria API - New to JPA 2.0, the Criteria API provides a typesafe API to perform object based queries. Used in-place of the JPQL.

A Native Query Example

Native Queries are easier to use because the do not require developers to learn the Java Persistence Query Language or Criteria API. Native Query usage should be kept at a minimum because the resulting code becomes tightly coupled with the underlying database structure.

Code 9-12 Native Query Example

```
Query query = em.createNativeQuery("SELECT * FROM Customer",
Customer.class);
List customers = query.getResultList();

Query query = em.createNativeQuery("SELECT * FROM SHARES WHERE SSN = '" +
customerId + "'", CustomerShare.class);
List shares = query.getResultList();
```

Summary

The following topics were presented in this module:

- The role of entity components
- The object-oriented data view provided by the Persistence API
- The purpose of a primary key
- Field and property-based entity classes
- The relationship of a persistence unit, persistence context, and entity manager
- The life cycle of entity components

Module 10

Implementing a Transaction Policy

Objectives

Upon completion of this module, you should be able to:

- Describe transaction semantics
- Compare programmatic and declarative transaction scoping
- Use the Java Transaction API (JTA) to scope transactions programmatically
- Implement a container-managed transaction policy
- Support optimistic locking with the versioning of entity components
- Support pessimistic locking with the `EntityManager` locking APIs
- Describe the effect of exceptions on transaction state
- Use JPA Transactions in a Java SE environment

Additional Resources

Additional resources – The following references provide additional information on the topics described in this module:

- Eric Jendrock, Debbie Carson, Ian Evans, Devika Gollapudi, Kim Haase, Chinmayee Srivathsa. “The Java EE 6 Tutorial,” [`http://java.sun.com/javaee/6/docs/tutorial/doc/`], accessed 1 August 2009.
- "Java Platform, Enterprise Edition 6 (Java EE 6) Specification", [`http://jcp.org/en/jsr/detail?id=316`], accessed 1 August 2009.
- "JSR 318: Enterprise JavaBeans, Version 3.1", [`http://www.jcp.org/en/jsr/detail?id=318`], accessed 1 August 2009.
- "JSR 317: Java Persistence, Version 2.0, Java Persistence API", [`http://jcp.org/en/jsr/detail?id=317`], accessed 1 August 2009.

Transaction Semantics

The concept of transactionality is important in the Java EE model. Not only are relational databases transactional, but in a complex system, a transaction might span multiple, distributed data sources of different types.

This course does not provide detailed descriptions for either the theory of transactions or the technicalities of transaction management. However, the Java EE transaction model abstracts these details, which leaves the developer to work with a relatively simple API or declarative transaction model. Behind the scenes, the application server's transaction management infrastructure might be complex, because it deals with two-phase commit, resource enlistment, and so on.

One of the design goals of the Java EE model was to provide a way for application development to be independent of these technicalities. Thus, the code is the same whether the application runs against a simple embedded database or a collection of distributed, synchronized enterprise information systems.

For the purposes of building Java EE applications, the transaction management concepts that the developer needs to understand are *atomicity*, *locking and isolation*, and the *flat transaction model*.

Atomicity

A transaction is *atomic* if it consists of a number of operations that must succeed or fail as a unit. For example, Figure 10-1 shows the process of transferring money between two bank accounts held on relational databases.

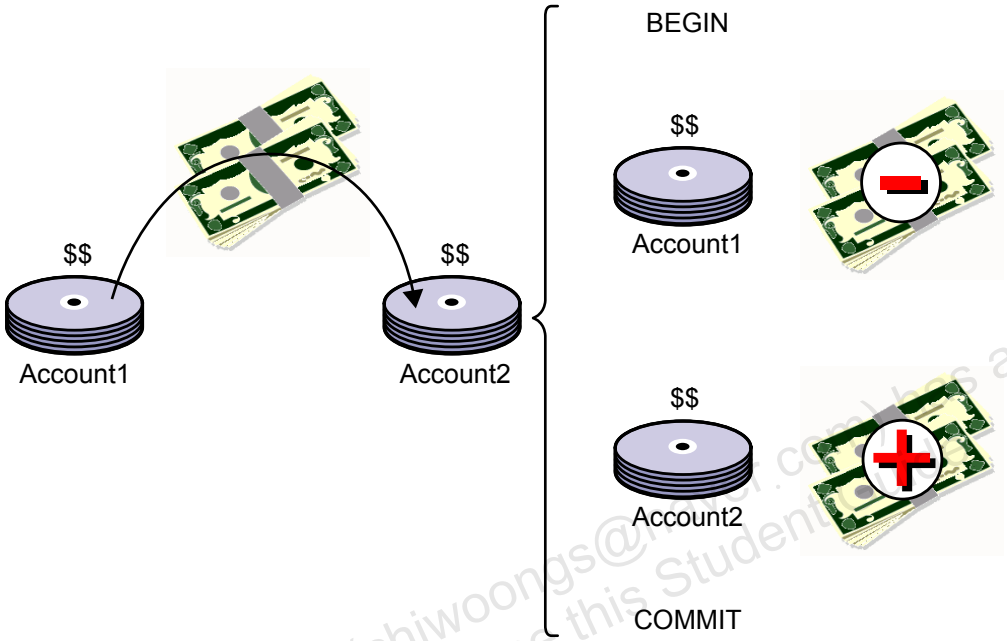


Figure 10-1 Example of Atomic Unit of Work

Because relational databases do not have a concept of *transferring money*, this operation must be completed in two steps:

- Reduce the amount of money in the account named `Account1`
- Increase the amount of money in the account named `Account2`

If both of these operations succeed, then there is no cause for concern. If both operations fail, then there is cause for concern, but the data is left in a consistent state. If one operation succeeds and the other operation fails, then there is likely to be trouble.

A transaction's *scope* extends from the start of its first operation, called the *begin point*, to the end of its last operation, called the *commit point*. If any operation between these two points fails, then the system must be restored to the state it had immediately before the begin point. Internally, the database engine uses logs to store the original state and monitor the success of the operations. Each change that is made during a transaction is logged, and if any operation fails, then the logged operations are undone from the end of the log back to the beginning of the log. This process is called *rolling back*.

In practice, few transactions are as simple as the transferring money example. Consider that the operation to reduce the amount of money in `Account1` is not itself atomic. That is, this first step in transferring money probably consists of a number of steps that are implemented in the application and a number of operations on the database. Consequently, the concept of scope is important because when you define a transaction in terms of its scope, rather than in terms of the number of operations it carries out, you get a much more straightforward programming model.

In addition, in a complex transaction, not every operation that is carried out along the way is a critical one. For example, if the transaction consists of ten operations, one of which is to send an email notification to the user, it is not necessarily the case that a failure of this notification should result in a failure of the entire transaction. Regardless of whether a particular type of failure should cause a transaction rollback is a design issue, which must be given effect in the implementation.

Locking and Isolation

Locking allows the database to handle multiple concurrent transactions, while still ensuring that integrity is maintained. With locking, after a particular transaction has carried out an update on a particular data item, other transactions are prohibited from updating or even reading that item again until the first transaction commits. Subsequent transactions are then forced to wait, and they are said to be *blocked* by the database. This process is called *transaction isolation*.

Figure 10-2 shows an example of locking in a transaction.

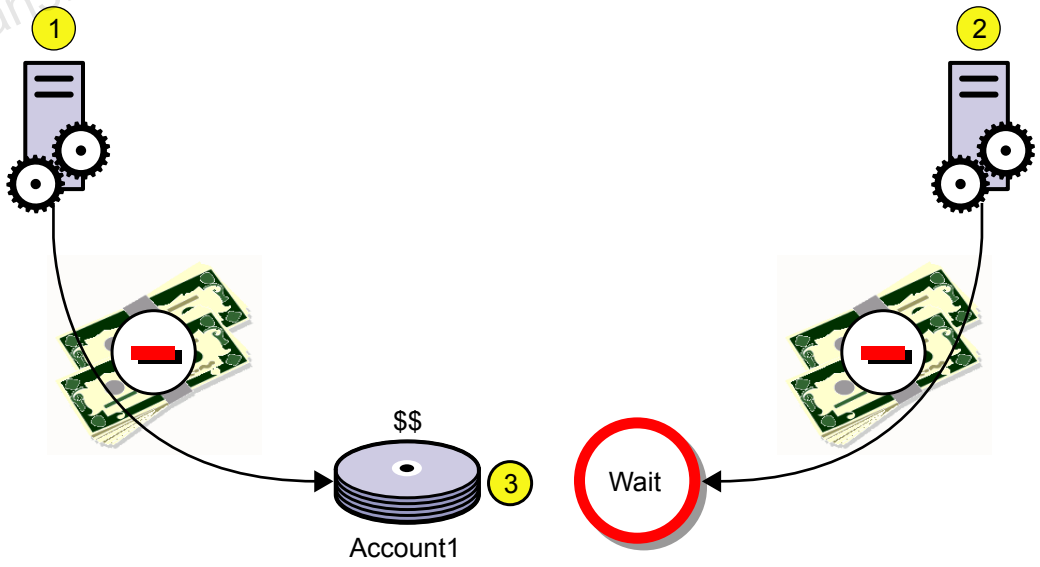


Figure 10-2 Effect of Locking on Multiple Accesses to the Same Data

The transaction shown in Figure 10-2 has the following three steps:

- The first transaction updates `Account1` in the database. While the first transaction is in process, `Account1` is locked.
- The second transaction attempts to update `Account1`, but cannot complete the update because the account is locked.
- When the first transaction completes, the lock on `Account1` is released, which allows the second transaction to continue.

Transaction Models

Different database implementations support a variety of transaction management models. In general, transaction management models can be divided into three types:

- Nested transaction model – A transaction is made up of subtransactions that run in parallel.

  If the main transaction fails, the subtransactions are rolled back. However, a failure in a subtransaction does not necessarily roll back other subtransactions. For example, if transaction 1 creates transaction 2, then in a nested transaction model, a failure in transaction 1 rolls back transaction 1 and transaction 2. However, a failure in transaction 2 does not automatically role back transaction 1, because transaction 2 is subordinate to transaction 1.

- Chained transaction model – A transaction is made up of subtransactions that run in sequence.

  If the main transaction fails, the subtransactions are rolled back. However, a failure in a subtransaction only causes a rollback to the start of that subtransaction. For example, if transaction 1 creates transaction 2, and transaction 2 creates transaction 3, then in a chained model, a failure in transaction 3 only rolls back transaction 3. A failure in transaction 2 rolls back transaction 2 and transaction 3. A failure in transaction 1 will rolls back all three transactions.

- Flat transaction model - A transaction *cannot* be made up of subtransactions.

  In a particular thread, only one transaction is in effect at a time.

The Java EE specification only supports the flat transaction model because this model is universally supported by database vendors. In addition, the flat transaction model leads to a simple API for transaction scoping. When a component issues a method call to commit a transaction, the component does not have to specify which transactions to commit because only one transaction can be in effect at a time. Flat transactions also allow for a simple declarative transaction model. More complex transaction models can be simulated in code if necessary.

Comparison of Programmatic and Declarative Transactions

It is important to remember that the developer of a Java EE application is only concerned with *scoping* transactions. Scoping refers to the selection of which operations the developer wants to group into a transaction. An alternative term is *transaction boundary demarcation*. The developer is never concerned with the technicalities of transaction coordination. That is, matters, such as the resource enlistment and log management, are the responsibility of the application server.

For some component types, the Java EE application developer can decide whether to use programmatic transaction scoping or declarative transaction scoping. Some component types only allow one method or the other. Where both methods are allowed, the decision about which method to use operates at the component level. For example, a particular EJB component cannot use both declarative and programmatic transaction scoping.

The following sections explore the similarities and differences between the programmatic and declarative transaction scoping methods.

Programmatic Transaction Scoping

With programmatic transaction scoping, you use JTA calls to determine which operations form a single transaction. This technique is often referred to as *bean-managed transactions* (BMT), but remember that the bean cannot actually manage transactions, it can only scope them.

Programmatic transactions can be used in servlets, JSP components, session beans, and message-driven beans.

Declarative Transaction Scoping

With declarative transaction scoping, you can set up transaction attributes for each method of each EJB component. Transaction attributes can be specified in the deployment descriptor or with annotations in the bean class. The EJB component's object then interact with the transaction infrastructure in accordance with these attributes. This technique is often referred to as a *container-managed transaction* (CMT).

Declarative transactions can be used in session beans, and message-driven beans, but cannot be used in servlets or JSP components.

The declarative transaction method is usually preferred over programmatic scoping, and the reasons for this preference are detailed in the following sections.

Programmatic Scoping as Opposed to Reusability

Programmatic transaction scoping is not preferred for EJB components for two main reasons:

- Transaction scope cannot be changed at assembly time.
- The container cannot create a single transaction that encompasses a mixture of programmatic and declarative transaction scoping.

Declarative scoping avoids these problems, and is therefore preferred. In addition, the use of JTA transactions in web components can usually be avoided in applications that use EJB components. Rather than having the web component call a number of methods in a JTA transaction, have the web component call a single method in an EJB component that in turn groups these methods into a transaction.

Developers who are new to the Java EE platform frequently ask how JTA transactions are different from Java™ DataBase Connectivity (JDBC™) transactions, and whether they can interoperate. JDBC transactions have connection scope, which means that they operate against a single back end and they commit when a connection is closed or committed.

A JTA transaction is scoped by its `begin()` and `end()` calls, and is not affected by opening and closing connections. Therefore, JTA transactions work against any number of back ends, and do not require a connection to be held open for an extended duration.

While you can use JDBC transactions in application servers, provided their limitations are accepted, there is seldom a good reason to do so. JTA is just as easy to use and, in a modern application server, just as efficient. In the *old* days, the use of JTA would imply a two-phase commit on every transaction, even if there was only one back end. Modern products recognize this and silently convert JTA transactions into JDBC transactions internally.

Using JTA to Scope Transactions Programmatically

JTA offers an abstraction of a transaction management infrastructure. Although the JTA specification is quite complex, most of the information is relevant to developers of application servers and database drivers and is not relevant to application developers. Application developers are only concerned with the `UserTransaction` interface, which is used for all transaction scoping operations in Java EE application components.

Getting a Reference to the `UserTransaction` Interface

You can use the JNDI API to get a reference to the `UserTransaction` interface, as shown in the following example:

Code 10-1 JNDI Lookup of a `UserTransaction`

```
import javax.transaction.UserTransaction;

InitialContext ic = new InitialContext();
UserTransaction ut = (UserTransaction)
  ic.lookup("java:comp/UserTransaction");
```

An EJB component, servlet, or JSP page can use annotations to obtain a reference to the `UserTransaction` interface.

Code 10-2 Dependency Injection of a `UserTransaction`

```
import javax.ejb.*;
import javax.annotation.*;
import javax.transaction.*;

@Stateful
public class MySessionBean implements MySession {

@Resource UserTransaction ut;

}
```

Using the `begin`, `commit`, and `rollback` Methods

Most uses of JTA to scope a transaction use the `begin`, `commit`, and `rollback` methods in the basic structure that is shown in Code 10-3.

Code 10-3 Using the `begin`, `commit`, and `rollback` Methods

```
import javax.ejb.*;
import javax.annotation.*;
import javax.transaction.*;

@Stateful
@TransactionManagement(BEAN)
public class MySessionBean implements MySession {
@Resource UserTransaction ut;

public void method() {
   try {
      ut.begin(); // Transaction starts here
      //... transaction operation 1,2,3
      ut.commit(); // Transaction finishes here

   } catch (Exception e) {
      ut.rollback(); // Oops: roll back
   }
}
}
```

Implementing a Container-Managed Transaction Policy

Declarative or container-managed transaction scoping is the technique of choice in EJB components. Declarative transaction scoping is portable, easy to apply, and does not limit the reusability of EJB components. Annotations are the preferred way to implement declarative scoping, but the transaction scope is also affected by the number and sequence of method calls.

Code 10-4 Using Declarative Transactions With Annotations

```
import javax.ejb.*;

@TransactionAttribute(TransactionAttributeType.REQUIRED)
public void someMethod () {...}
```

Note – The interaction between the containers, transaction infrastructure, and deployment descriptor might seem complex. It might help to keep in mind that in many cases you are required to do nothing. All of the methods of CMT EJB components have a default of the `REQUIRED` transaction attribute.

Container Interactions With the Transaction Management Infrastructure

With declarative transactions, the container makes adjustments to the transaction state on entry to, and exit from, every method in every EJB component. The container makes these adjustments in accordance with a set of transaction attributes that are defined in the deployment descriptor.

Figure 10-3 shows the basic sequence that is used to define the transaction attributes.

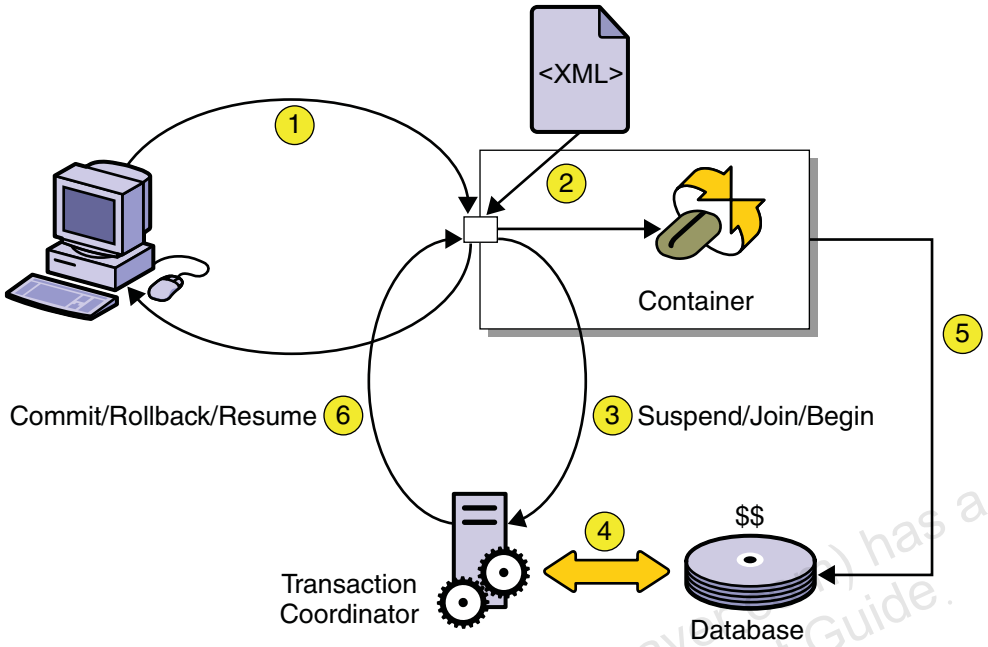


Figure 10-3 Container Interactions With the Transaction Management Infrastructure

Figure 10-3 illustrates the following basic sequence:

1. The client of the EJB component calls a business method. The client calls the business method by way of the component object. At this point, either a transaction is in progress, or it is not. Because the transaction model is flat, the container does not have to be concerned about different transactions in the same thread of execution.
2. The container obtains the transaction attribute for this method from the information that is supplied in the deployment descriptor. There are six possible attributes, which are described in Table 10-1 on page 10-16.
3. Depending on whether a transaction was in progress, and also on the method's transaction attribute, the container adjusts the transaction state. It can either suspend the current transaction, join the new method onto the current transaction, or begin an entirely new transaction, which first suspends the original transaction. A transaction that is suspended is *on hold* for the duration of the method call. It will not be rolled back if the method fails, and the suspended transaction can be resumed later.

4. The transaction manager does whatever is necessary on the various databases and other information sources to affect the container's decision. The transaction manager also sets up internal data structures in such a way that later access to databases brings about the correct transactional operations.
5. The container calls the business method on the EJB component and the container might then read or update the database.
6. On exit from the method, the container completes whatever it started on entry. For example, if it began a new transaction, the container attempts to commit the transaction. If the container suspended the existing transaction, the container resumes the transaction. The container can also roll back a transaction, either because the commit operation fails or because the rollback flag has been set by a business method within the scope of a transaction.

Controlling the Container's Behavior Using Transaction Attributes

The previous section described how the container modified the transaction state on entry to, and exit from, each method. Table 10-1 explains how the container uses the transaction attributes supplied by the developer or assembler to decide how to modify a transaction.

Table 10-1 Transaction Attributes and Effects

Attribute	Effect
`Required`	The method becomes part of the caller's transaction. If the caller does not have a transaction, the method runs in its own transaction. This attribute gives acceptable results in most cases because it has the effect that the transaction scope extends from the start to the end of the client's method call on the business logic of the EJB components, provided that the client has not begun a transaction. Any failures within this scope cause the whole transaction to roll back.
`RequiresNew`	The method always runs in its own transaction. Any existing transaction is suspended. Use this attribute when the method carries out operations that are considered a transaction, but when the caller's transaction should not be affected in the event of a failure in this method.
`NotSupported`	The method never runs in a transaction. Any existing transaction is suspended. Use this attribute for methods that are ancillary to the transaction. For example, use the `NotSupported` method for transactions, such as logging, debugging, and notification.
`Supports`	The method becomes part of the caller's transaction if there is one. If the caller does not have a transaction, the method does not run in a transaction. In practice, this attribute is rarely used.
`Mandatory`	It is an error to call this method outside of a transaction. In practice, this attribute is rarely used.
`Never`	It is an error to call this method in a transaction. In practice, this attribute is rarely used.

Transaction Scope and Application Performance

As it does in other programming models, transaction scope has a significant effect on application throughput in Java EE applications. However, with the Java EE model, you must balance the additional factor of the effect of transaction scope on database synchronization. Of course, the business rules of the application might themselves dictate the transaction scope. In the example of transferring money from one bank account to another bank account, it is clear that both operations must be part of the same transaction. No improvement in efficiency is likely to outweigh the risk of data corruption.

It takes considerable experience to be able to make the necessary design decisions to get good throughput and avoid compromising the business rules. In this course, only a brief outline can be given.

Transaction Scope and Entity Synchronization

Entity components inherit the transactional state of a calling component, such as a session bean. This means that you do not add any transactional attributes, such as `Required`, to entity classes. Long-term locking can be used in JPA 2.0. In place of long-term record locking, entity classes can be configured to use optimistic locking in any version of JPA. It is also assumed that the database isolation level configured for entity components will be read-committed, because only committed data can be read in a transaction.

Optimistic locking in the Java Persistence API specification uses a version tracking ability to keep track of the revision of entity components. If you attempt to modify an entity component that uses optimistic locking and the entity's persistent data was modified by a concurrently running transaction, then the current transaction will fail.

Optimistic Locking

Enterprise developers must concern themselves with scalability and concurrency while maintaining data integrity. A standard approach to maintaining database integrity with many concurrent users that access shared data is to use record locking. Locking rows or tables for long periods of time to keep multiple records from being updated by multiple users is referred to as pessimistic locking.

A basic implementation of pessimistic locking would lock data when it is read and not unlock until all modifications have taken place. Pessimistic locking does a good job of ensuring predictable database updates. However, a potential issue with long-term record locking using pessimistic locking is one of scalability. The more concurrent users there are of a database application, the more likely it is that those users will request the same data causing contention.

Optimistic locking tries to avoid acquiring long-term data locks. Data can be locked during a read and during an update, but would not be locked in-between a read and update regardless of the duration of the edit. This locking model scales better with a large amount of concurrent users accessing the same data especially when database operations consist mostly of reads. A drawback to optimistic locking is that it becomes easy to over-write another user's changes.

Note – Transactions do not always lock database rows. A transaction simply means that several operations will either commit or fail as a unit. If one operation fails all transacted operations in the same transaction should fail. Without additional work there is very little that will cause a JPA update to fail, even concurrent transactions updating the same records in a database will complete successfully.

Versioning

The Java Persistence API specification outlines the addition of version numbers to entity data to avoid over-writing other data updates accidentally. This *versioning* ability is only used when developers enable it.

When an entity is retrieved from a database its version number is also looked up. If the entity is modified and committed to the database, the in- memory version number is compared with the version currently stored in the database. If the version numbers match, the update is allowed and the version number in the database is automatically incremented. If the version numbers did not match, indicating another user has already modified the data, an exception is thrown.

A basic implementation of versioning can use an integer database column in the same table used for entity data. Data should not be written to the field or property used for the version.

Code 10-5 A Versioning Field in an Entity Class

```
@Version @Column(name = "VERSION")
private int version;

public int getVersion() {
   return version;
}
```

Note – When an entity is merged and its version does not match the stored version, the merge fails with a `javax.persistence.OptimisticLockException`.

Pessimistic Locking

Optimistic locking with versioning does not work well when there is high write contention. In the case of an `OptimisticLockException` the developer must write code to reattempt the operation until successful.

Note – Many JPA 1.0 providers have proprietary methods to enable pessimistic locking in some form. Code that uses pessimistic locking with a JPA 1.0 provider will not be 100% portable.

Methods for performing pessimistic locking are part of the `EntityManger` class. Additional pessimistic locking capabilities are included in the `Query` and `NamedQuery` features of JPA which are beyond the scope of this course. The Java Persistence API 2.0 specification does not specify the mechanism that will be used to lock the records in the underlying database.

Basic usage involves specifying a lock mode of `javax.persistence.LockModeType.PESSIMISTIC` as an argument to various methods of the `EntityManager` class.

Code 10-6 Method syntax of the overloaded `EntityManager.find` method used to retrieve an entity instance with a pessimistic lock.

```
<T> T find(java.lang.Class<T> entityClass, java.lang.Object primaryKey,
LockModeType lockMode)
```

Code 10-7 Method syntax of the `EntityManager.lock` method used to lock an entity instance already in the `PersistenceContext`.

```
void lock(java.lang.Object entity, LockModeType lockMode)
```

If the lock can not be obtained and the current transaction will be marked for rollback a `PessimisticLockException` will be thrown. When the lock can not be obtained but the current transaction does not require a rollback a `LockTimeoutException` will be thrown.

Effect of Exceptions on Transaction State

With programmatic transaction scoping, you are responsible for handling exceptions and, if necessary, failing a transaction by calling the `UserTransaction.rollback` method. This section describes the effect of exceptions on transactions that are managed by the EJB component's container.

Runtime Exceptions and Rollback Behavior

In general, with declarative transaction scoping, the container rolls back the current transaction if it catches a runtime or unchecked exception. For example, errors that result in a runtime exception being thrown cause the transaction to fail. This includes `NullPointerException`, for example. If the EJB component detects that a transaction should fail, perhaps as the result of some logical error, then it usually throws an `EJBException`, which is a runtime exception. You need to be aware of this, and decide whether, in a particular case, this is the desired behavior. In most cases it is.

Throwing a checked exception from a business method never, in itself, causes a rollback. This is true no matter how catastrophic the failure might be. However, the failure that caused the exception might itself have caused a rollback.

For example, consider a session bean that uses an entity class to insert a new data item. Suppose the `merge` operation failed and threw an `OptimisticLockException`. This example is an unchecked exception, so throwing it from the session bean to its client does roll back the current transaction. Most Java Persistence API-caused exceptions are unchecked exceptions because they inherit from `RuntimeException`. You should catch any subclasses of `PersistenceException` when using the Java Persistence API within session beans to keep the current transaction from failing, if desired.

Using the `EJBContext` Object to Check and Control Transaction State

When you use declarative transaction scoping, transaction failures are under the control of the container. An EJB component might need to determine whether a transaction has failed somewhere other than in the current method. The usual reason for making this determination is to avoid carrying out operations that are doomed to fail later. If the current transaction has already failed at an earlier point, perhaps as a result of a database error, then there is no benefit in completing further work in the current method, because the container will only have to roll it back on exit.

The `EJBContext` object that is injected in EJB components at runtime provides two methods that are useful in this respect. The `getRollbackOnly` method returns true if the transaction has already failed. Typically, an EJB component calls this before embarking on a lengthy database operation.

You can use the `setRollbackOnly` method to fail a container-managed transaction without throwing a runtime exception. This method sets the rollback flag, which might have the effect of making the transaction roll back immediately, or of causing the container to roll it back later. In either case, the transaction fails and can be rolled back at some point. The decision to set the rollback flag cannot be revoked later in the transaction.

JPA Transactions in Java SE applications

JTA based entity managers are supported in an EJB environment including servers supporting the Java EE web profile. In a Java SE environment JTA will not typically be available and a resource-local entity manager will be used. When using a resource-local entity manager begins, commits and rollbacks become the responsibility of the application developer.

A resource-local entity manager is specified in the `persistence.xml` configuration file. For a complete example see "An example `persistence.xml` file for a Java SE JPA application." on page 9-26.

Code 10-8 Resource-local entity manager selection in `persistence.xml`

```
<persistence-unit name="StockPU" transaction-type="RESOURCE_LOCAL">
...
</persistence>
```

The `EntityTransaction` API controls controls transactions in an application using a resource-local entity manager.

Code 10-9 Programmatically obtaining an `EntityManager` and manually controlling transactions.

```
EntityManagerFactory emf =
Persistence.createEntityManagerFactory("StockPU");
EntityManager em = emf.createEntityManager();
EntityTransaction entityTransaction = entityManager.getTransaction();
entityTransaction.begin();
// ...
entityTransaction.commit();
```

Summary

Although transaction management infrastructure can be complex, particularly in a distributed application, a transaction is logically straightforward. A transaction begins, and then ends. Everything between these two demarcation points is part of that transaction.

The transaction model of the Java EE platform exposes this simple logic to the developer, while concealing the implementation. You can demarcate transactions using API calls, or using the declarative model in EJB components. Because EJB components have access to this declarative model, modern practice favors the concentration of transactional logic in EJB components, rather than in servlets or in other clients.

Module 11

Developing Java EE Applications Using Messaging

Objectives

Upon completion of this module, you should be able to:

- Describe JMS technology
- Write a message producer
- Write a queue message browser
- List the capabilities and limitations of enterprise components as messaging clients

Additional Resources

Additional resources – The following references provide additional information on the topics described in this module:

- Eric Jendrock, Debbie Carson, Ian Evans, Devika Gollapudi, Kim Haase, Chinmayee Srivathsa. “The Java EE 6 Tutorial,” [`http://java.sun.com/javaee/6/docs/tutorial/doc/`], accessed 1 August 2009.
- "JSR 914: Java Message Service (JMS) API, Version 1.1", [`http://www.jcp.org/en/jsr/detail?id=914`], accessed 1 August 2009.

JMS API Technology

This section provides an overview of JMS API technology. Figure 11-1 shows the relationship among the main participants of the JMS API messaging system.

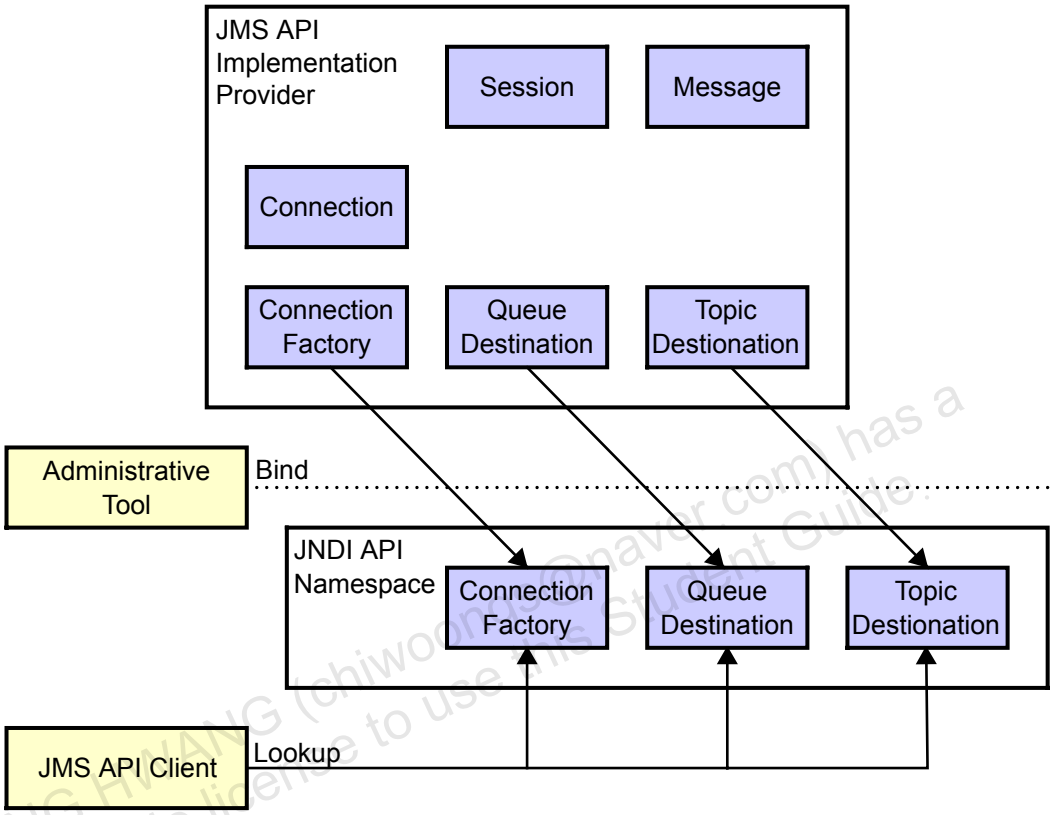


Figure 11-1 Messaging System Participants

The JMS API consists of a set of interfaces and classes. A JMS technology provider (JMS provider) is a messaging system that provides an implementation of the JMS API. For an application server to support JMS technology, you must place the administered objects (connection factories, queue destinations, and topic destinations) in the JNDI technology namespace of the application server. You use the administrative tool supplied by the application server to perform this task.

Administered Objects

Administered objects are objects that are configured administratively (as opposed to programmatically) for each messaging application. A JMS provider supplies the administered objects. An application server administrator or application deployer configures these objects and places them in the JNDI technology namespace.

The two types of administered objects are destinations and connection factories.

Destinations are message distribution points. Destinations receive, hold, and distribute messages. Destinations fall into two categories: queue and topic destinations:

- Queue destinations implement the point-to-point messaging protocol.
- Topic destinations implement the publish/subscribe messaging protocol.

Connection factories are used by a JMS API client (JMS client) to create a connection to a JMS API destination (JMS destination). Connection factories fall into two categories: queue and topic factories.

Figure 11-2 shows both types of administered objects supplied as part of a JMS provider.

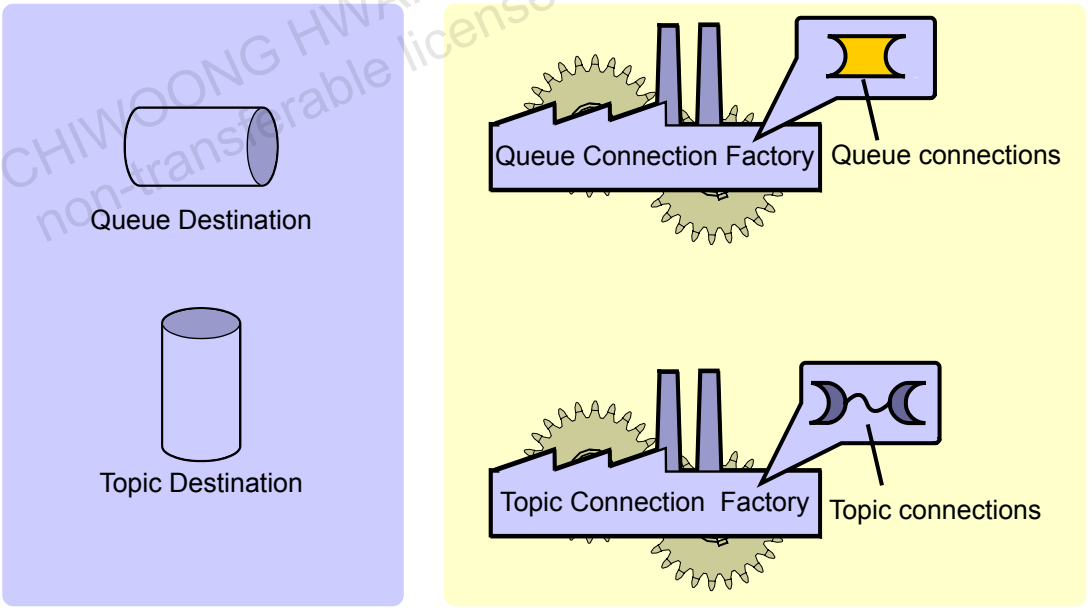


Figure 11-2 Administered Objects

Messaging Clients

Clients are applications that produce or consume messages. Message producer clients create and send messages to destinations.

Message consumer clients consume messages from destinations. Message consumers can be either asynchronous message consumers or synchronous message consumers.

Asynchronous consumer clients register with the message destination. When a destination receives a message, it notifies the asynchronous consumer, which then collects the message.

Synchronous message consumer clients collect messages from the destination. If the destination is empty, the client blocks until a message arrives.

Figure 11-3 shows the three types of messaging clients.

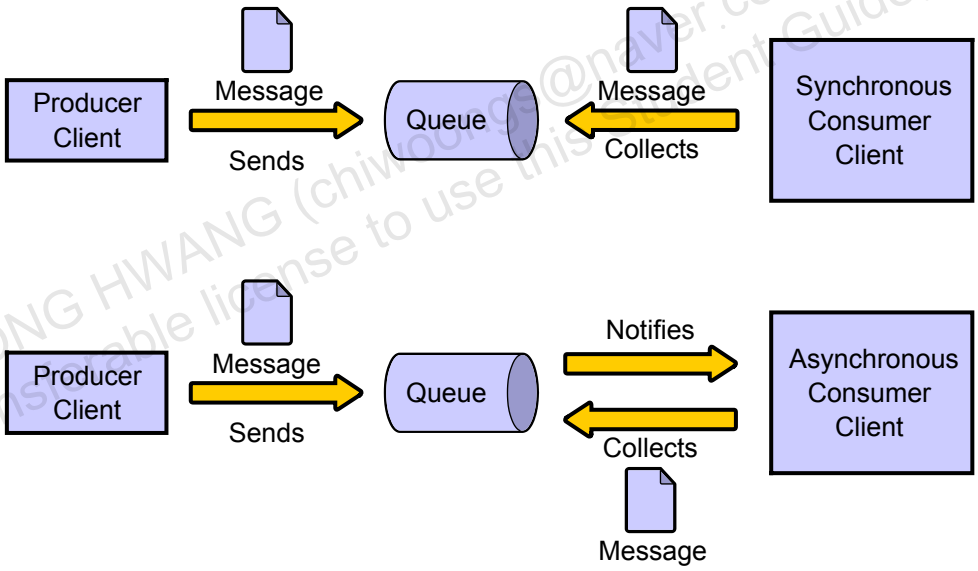


Figure 11-3 Messaging Clients

Messages

Messages are objects that encapsulate application information created by a message producer. Messages are sent to a destination that distributes them to message consumers.

Figure 11-4 shows the three parts of a JMS API message.

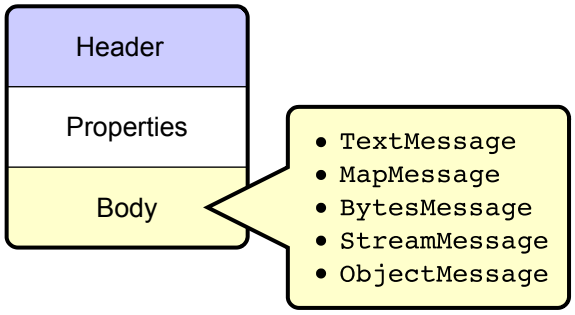


Figure 11-4 Message Construction and Types

The following three characteristics apply to messages:

- The header contains standard information for identifying and routing the message.
- Properties provide a mechanism for the classification of messages. Properties are name/value pairs. You can set or read the value of any property. You can customize the property list by adding a name/value pair. Properties allow destinations to filter messages based on property values. Properties also provide interoperability with some message service providers.
- The body contains the actual message in one of several standard formats.

Table 11-1 lists the JMS API message types.

Table 11-1 JMS Technology Message Types

Message Type	Contents of the Message Body
`TextMessage`	A `java.lang.String` object
`MapMessage`	A set of name-value pairs, for example, a hash table
`BytesMessage`	A stream of uninterpreted bytes or binary data
`StreamMessage`	A stream of Java technology primitive values filled and read sequentially
`ObjectMessage`	A serializable Java technology object

Point-to-Point Messaging Architecture

Figure 11-5 illustrates the point-to-point messaging architecture, which is based on the concept of a message queue. A message queue retains all messages until they are consumed. The distinguishing factor between the point-to-point and publish/subscribe architectures is that in most cases, each point-to-point message queue has only one message consumer, and there are no timing dependencies between the sender and receiver. That is, a receiver gets all the messages that were sent to the queue, even those sent before the creation of the receiver. The queue then deletes messages on the acknowledgement of successful processing from the message consumer.

Note – A message queue can have multiple consumers. When one consumer consumes a message from the queue, the message is marked as consumed. Other consumers cannot also consume the same message.

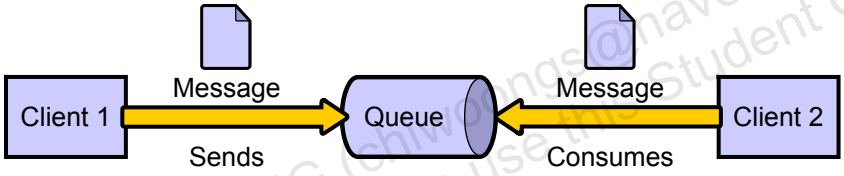


Figure 11-5 Point-to-Point Architecture

Publish/Subscribe Messaging Architecture

Figure 11-6 illustrates the publish/subscribe architecture, which is based on the concept of a message topic. A topic can have multiple consumers. Message producers publish messages to a topic. The topic retains messages until the messages are distributed to all consumers. There is a timing dependency between publishers and subscribers. That is, subscribers do not receive messages that were sent before their subscription to the topic or while the subscriber is inactive.

Note – The JMS API supports the concept of a durable subscriber. A topic retains all messages for subsequent delivery to inactive durable subscribers.

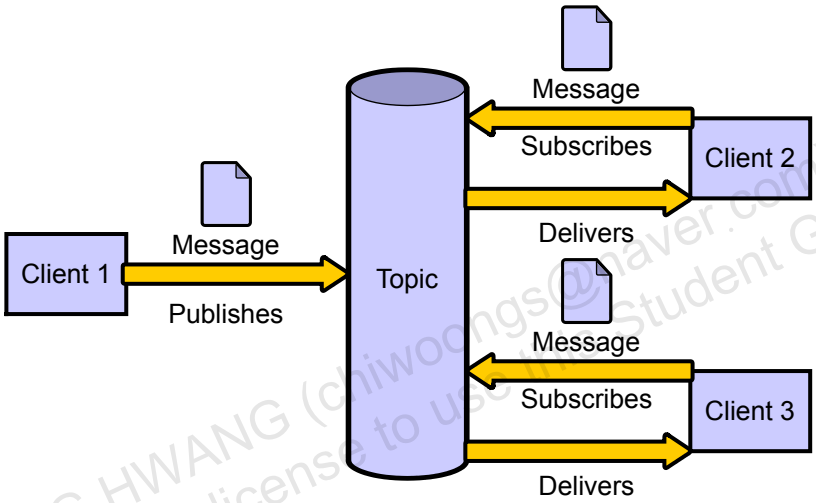


Figure 11-6 Publish/Subscribe Architecture

Creating a Queue Message Producer

Figure 11-7 illustrates the steps required to create a queue message producer.

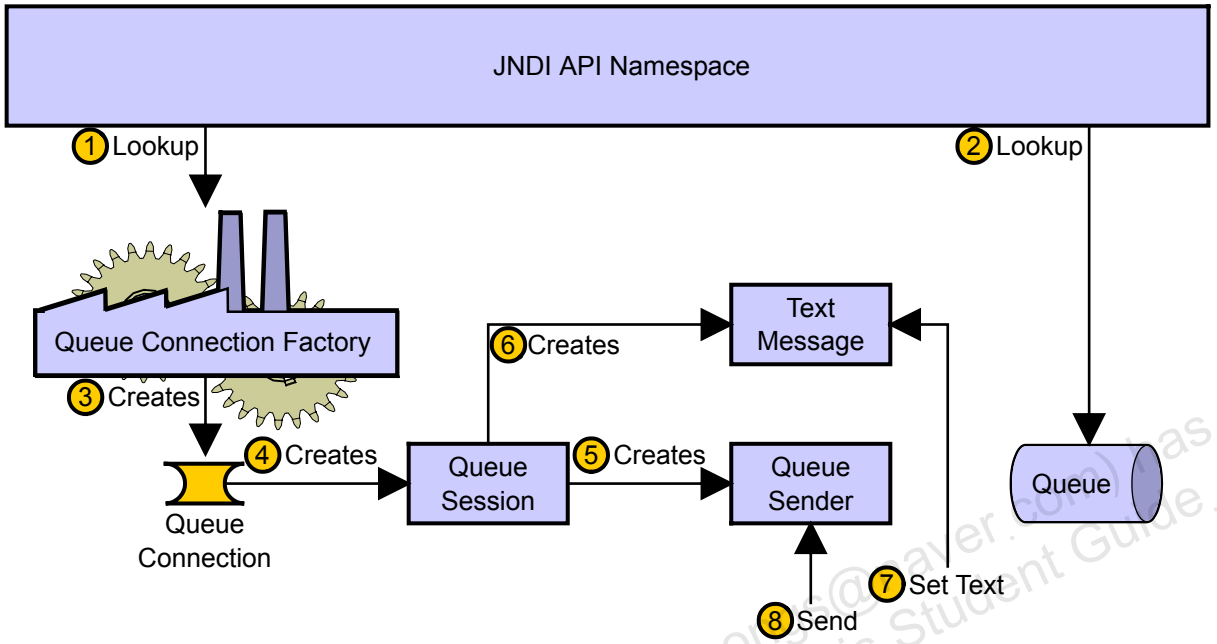


Figure 11-7 Creating a Queue Message Producer

The following steps explain how you can create a message queue producer.

1. Obtain a connection factory using injection or the JNDI API.

   A connection factory is an administered object. An instance of the `QueueConnectionFactory` class is required. You use it to create the `Connection` object in Step 3.

2. Obtain the message queue using injection or the JNDI API.

   A message queue is an administered object. A message queue implements the point-to-point message protocol destination.

3. Create a `Connection` object using the connection factory.

   A connection encapsulates a virtual connection with a JMS API provider enabling messaging clients to send messages and to receive messages from destinations.

4. Create a `Session` object using the connection.

   A `Session` object provides a single-threaded context for producing and consuming messages.

5. Create a `QueueSender` or `MessageProducer` object using the `Session` object.

   A `QueueSender` object provides the API to send messages to a queue destination.

6. Create one or more `Message` objects using the `Session` object.

   Use the session object to create the required type of message object. After creation, populate the message with the required data.

7. (Optional) Populate the message with text using the `setText` method of the `TextMessage` object.

   - A `TextMessage` can be populated when it is created.

8. Send one or more `Message` objects using the `QueueSender` or `MessageProducer` object.

   Use the `send` method of the `QueueSender` or `MessageProducer` to send the messages to the queue destinations.

Message Producer Code Example

Code 11-1 Example Code for a JMS Message Producer

```
package com.example;

import java.io.IOException;
import javax.annotation.Resource;
import javax.jms.*;
import javax.servlet.*;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.*;

@WebServlet("/JMSAddServlet")
public class JMSAddServlet extends HttpServlet {

    @Resource(mappedName = "jms/QueueConnectionFactory")
    javax.jms.QueueConnectionFactory queueConnectionFactory;
    @Resource(mappedName = "jms/DemoQueue")
    javax.jms.Queue queue;

    protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {

        String text = request.getParameter("text");
        try {
            Connection connection =
queueConnectionFactory.createConnection();
            Session session = connection.createSession(false,
Session.AUTO_ACKNOWLEDGE);

            TextMessage message = session.createTextMessage(text);
            MessageProducer producer = session.createProducer(queue);
            producer.send(message);

            producer.close();
            session.close();
            connection.close();

            response.sendRedirect("JMSListServlet");

        } catch (JMSException ex) {
            request.setAttribute("exception", ex);
```

```
            RequestDispatcher dispatcher =
request.getRequestDispatcher("jms_error.jsp");
            dispatcher.forward(request, response);
        }
    }

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
response)
    throws ServletException, IOException {
        processRequest(request, response);
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse
response)
    throws ServletException, IOException {
        processRequest(request, response);
    }

}
```

Note – A servlet, session bean or message-driven bean can use resource injection services of the container to obtain references to administered objects, such as a `QueueConnectionFactory` or a `Queue` or a `Topic`. Non-managed objects may use JNDI to obtain references to JMS administered objects.

Code 11-2 Using JNDI to obtain JMS administered objects

```
try {
    QueueConnectionFactory queueConnectionFactory =
(QueueConnectionFactory)jndiContext.lookup("QueueConnectionFactory");
    Queue queue = (Queue) jndiContext.lookup(queueName);
} catch (NamingException e) {
    //...
}
```

Queue Message Browser

A queue message browser allows the messages in a queue that have not been consumed to be viewed. Browsing is not supported for topics, does not consume any messages, and does not enable message editing or deleting.

Queue Message Browser Code Example

Code 11-3 Example Code for a Queue Message Browser

```
package com.example;

import java.io.IOException;
import java.util.*;
import javax.annotation.Resource;
import javax.jms.*;
import javax.servlet.*;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.*;

@WebServlet("/JMSListServlet")
public class JMSListServlet extends HttpServlet {

    @Resource(mappedName = "jms/QueueConnectionFactory")
    javax.jms.QueueConnectionFactory queueConnectionFactory;
    @Resource(mappedName = "jms/DemoQueue")
    javax.jms.Queue queue;

    protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {
        List<TextMessage> messages = new ArrayList<TextMessage>();
        try {
            Connection connection =
queueConnectionFactory.createConnection();
            Session session = connection.createSession(false,
Session.AUTO_ACKNOWLEDGE);

            QueueBrowser browser = session.createBrowser(queue);
            Enumeration queueEnum = browser.getEnumeration();
            while (queueEnum.hasMoreElements()) {
                Object obj = queueEnum.nextElement();
                if(obj instanceof TextMessage) {
```

```
                    messages.add((TextMessage)obj);
                }
            }
            browser.close();
            session.close();
            connection.close();

            request.setAttribute("messages", messages);
            RequestDispatcher dispatcher =
request.getRequestDispatcher("jms_list.jsp");
            dispatcher.forward(request, response);

        } catch (JMSException ex) {
            request.setAttribute("exception", ex);
            RequestDispatcher dispatcher =
request.getRequestDispatcher("jms_error.jsp");
            dispatcher.forward(request, response);
        }
    }

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        processRequest(request, response);
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        processRequest(request, response);
    }

}
```

Evaluating the Capabilities and Limitations of EJB Components as Messaging Clients

Table 11-2 shows the messaging capabilities of enterprise beans.

Table 11-2 Messaging Capabilities of Enterprise Beans

Enterprise Component	Producer	Synchronous Consumer	Asynchronous Consumer
Session bean	Yes	Not recommended	Not possible
Message-driven bean	Yes	Not recommended	Yes
Servlet	Yes*	Not recommended	Not possible

Using Enterprise Components as Message Producers

Session and message-driven beans are capable of becoming message producers. To add this capability, you include code to invoke the required methods of the JMS API. Servlets are able to act as message producers however session beans are more typically used.

Using EJB Components as Message Consumers

Synchronous message consumers block and tie up server resources. For this reason, you should not use servlets or session and message-driven beans as synchronous message consumers.

Message-driven beans consume messages asynchronously are the preferred method to create message consumers.

Summary

The JMS API provides an abstraction on top of a messaging system. The two messaging models are a point-to-point model using Queues and a publish-subscribe model using Topics. Just about any component can produce messages, but because of the strict thread management enforced by an application server, not all components are suited to consume messages.

Module 12

Developing Message-Driven Beans

Objectives

Upon completion of this module, you should be able to:

- Describe the properties and life cycle of message-driven beans
- Create a JMS message-driven bean
- Create lifecycle event handlers for a JMS message-driven bean

Additional Resources

Additional resources – The following references provide additional information on the topics described in this module:

- Eric Jendrock, Debbie Carson, Ian Evans, Devika Gollapudi, Kim Haase, Chinmayee Srivathsa. “The Java EE 6 Tutorial,” [`http://java.sun.com/javaee/6/docs/tutorial/doc/`], accessed 1 August 2009.
- "JSR 318: Enterprise JavaBeans, Version 3.1", [`http://www.jcp.org/en/jsr/detail?id=318`], accessed 1 August 2009.
- "JSR 914: Java Message Service (JMS) API, Version 1.1", [`http://www.jcp.org/en/jsr/detail?id=914`], accessed 1 August 2009.

Introducing Message-Driven Beans

Message-driven beans are designed to function as asynchronous message consumers. Message-driven beans implement a message listener interface. For example, JMS message-driven beans implement the JMS API `MessageListener` interface. The bean designer is responsible for the message listener method code (the `onMessage` method for JMS message-driven beans).

The deployer must supply the information for the container to register the message-driven bean as a message listener with the destination.

Java EE Technology Client View of Message-Driven Beans

Figure 12-1 shows the Java EE technology client view of a message-driven bean.

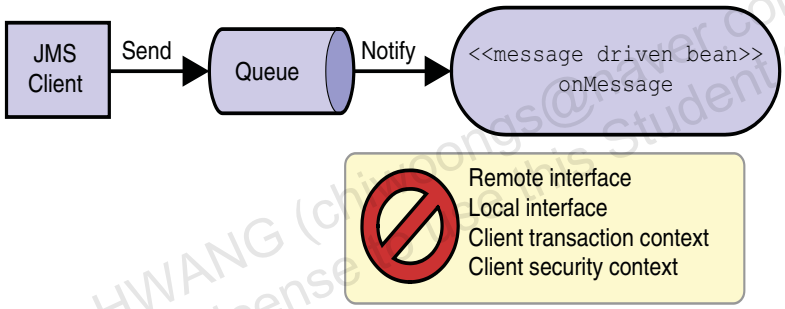


Figure 12-1 Java EE Technology Client View of a Message-Driven Bean

Message-driven beans do not have remote component, or local component interfaces.

Message-driven beans do not provide Java EE components with a direct client interface. Java EE technology clients communicate with a bean by sending a message to the destination (queue or topic) for which the bean is the `MessageListener` object.

Message-driven beans have no client-visible identity. They hold no client conversational state. They are anonymous to clients.

Message-driven beans are transaction aware but will not participate in a client's transaction because they have no client. The message sender's transaction and security contexts are unavailable to a message-driven bean. Consequently, message-driven beans must start their own transaction and security context.

Containers can create and pool multiple instances of message-driven beans to service the same message destination. The operation of these instances are independent of each other. When a message arrives at a message destination, the container chooses the next available instance associated with that destination to service the message.

Life Cycle of a Message-Driven Bean

Figure 12-2 shows the life cycle of a message-driven bean.

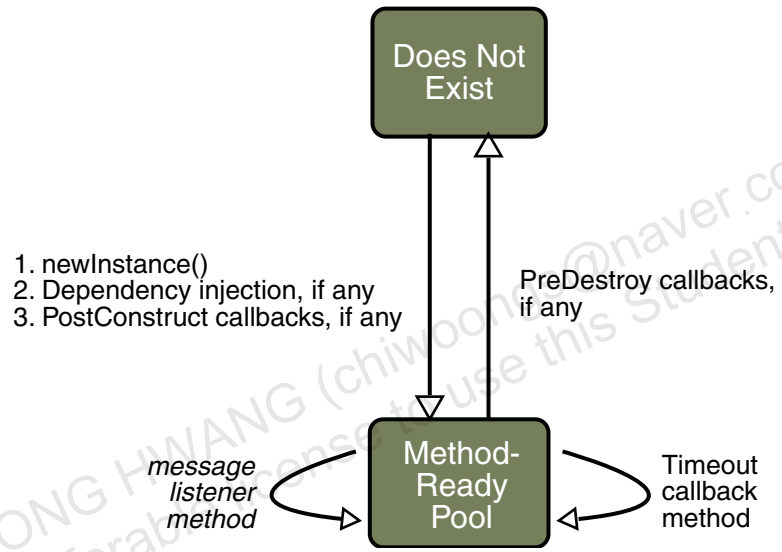


Figure 12-2 Message-Driven Bean Life Cycle

The container creates a message-driven bean by invoking the `newInstance` method. After the bean instance is created, the container, if applicable, injects the bean's `MessageDrivenContext` instance, and performs any other dependency injection, as specified by metadata annotations on the bean class or by the deployment descriptor. The container then calls the bean's `PostConstruct` callback method, if any. After which the container adds the bean to the method-ready pool. The bean is then ready to service the message notifications presented to it through the message listener method.

Types of Message-Driven Beans

In the EJB 3.0 specification EJB containers support the following two types of message-driven beans. The two types are distinguished by the message listener interface type they implement:

- JMS message-driven beans

  JMS message-driven beans are annotated with the `@javax.ejb.MessageDriven` annotation. JMS message-driven beans also implement the `onMessage` method of the `javax.jms.MessageListener` interface. The bean itself must implement the `javax.jms.MessageListener` interface or supply a property to the `@MessageDriven` annotation of `messageListenerInterface = javax.jms.MessageListener`.

  An application server can integrate the support for JMS message-driven beans directly into the container or use a supporting resource adapter based on the Java EE connector architecture.

- Non-JMS message-driven beans

  Non-JMS message-driven beans implement a message listener interface specific to the messaging service type that they support. In addition, non-JMS message-driven beans can optionally directly or indirectly implement the `javax.ejb.MessageDriven` interface.

  Non-JMS message-driven beans depend on a Java EE connector-based resource adapter based on the specific messaging service type.

Creating a JMS Message-Driven Bean

To create a message-driven bean class, complete the following steps:

1. Declare the message-driven bean class.

   The class must be public, but not final or abstract.

2. Annotate the message-driven bean class using the `MessageDriven` metadata annotation.

   The `MessageDriven` metadata annotation contains a number of attributes. The following list describes the most important of these attributes.

   - A property to the `@MessageDriven` annotation of:

   ```
   activationConfig =  {
   @ActivationConfigProperty(propertyName =
   "destinationType", propertyValue = "javax.jms.Queue")
   }
   ```

   - Use the `mappedName` attribute to specify the name of the message destination associated with the message-driven bean.

3. Optionally, use resource injection to obtain the `MessageDrivenContext` instance associated with the bean.

4. Include, in the message-driven bean class, a public no-argument constructor.

   Create or acquire any resources the `onMessage` method might require.

5. Write the `onMessage` method of the `MessageListener` interface.

   Supply the code to process the received message. This code might include:

   - Accessing entity instances and session beans executing in the same JVM machine using their local interfaces
   - Accessing session beans executing in another JVM machine using their remote interfaces
   - Sending messages to a message destination

6. Do not define a `finalize` method.

Code 12-1 shows a simple example of a message-driven bean class.

Code 12-1 Message-Driven Bean Example

```
1   package com.example;
2
3   import javax.ejb.*;
4   import javax.jms.*;
5
6   @MessageDriven(mappedName = "jms/DemoQueue", activationConfig = {
7       @ActivationConfigProperty(propertyName = "destinationType",
8       propertyValue = "javax.jms.Queue")
9   })
10  public class MDBExample implements MessageListener {
11
12      public void onMessage(Message message) {
13          System.out.print("Consumed message.");
14      }
15  }
```

Creating a JMS Message-Driven Bean: Adding Life-Cycle Event Handlers

With the EJB 3.0 specification, you are not required to provide life-cycle event handlers for message-driven beans. However, message-driven beans do generate life-cycle events and you can optionally provide event handlers for these methods.

Message-driven beans generate the following life-cycle events:

- `PostConstruct`

  The `PostConstruct` callback occurs after the construction of the bean instance but before the first message listener method invocation on the bean. This is at a point after which any dependency injection has been performed by the container.

- `PreDestroy`

  The `PreDestroy` callback occurs at the time the bean instance is removed from the pool or destroyed.

Defining a Callback Handler in the Bean Class

To define a callback handler in a message-driven bean class, complete the following steps:

1. Declare, in the bean class, a callback method with the following signature:
   `public void methodName()`
2. Ensure that the callback method conforms to the following rules:
   - It can have any type of access modifier (`public`, default, `protected`, or `private`).
   - It must not throw an application exception.
   - It can throw a runtime exception.
   - It must not use dependency injected values.
   - It must not rely on a specific transaction or security context.
3. Annotate the method with the appropriate life-cycle event handler metadata annotation. For a `PostConstruct` method, use the `PostConstruct` metadata annotation. For a `PreDestroy` method, use the `PreDestroy` metadata annotation.

Code 12-2 shows an example of life-cycle event handler methods `obtainResources` and `releaseResources` defined within a bean class.

Code 12-2 Callback Methods in a Bean Class

```
1   import javax.ejb.*;
2   import javax.jms.*;
3   import javax.annotation.*;
4
5   @MessageDriven (mappedName = "jms/DemoQueue", activationConfig = {
6       @ActivationConfigProperty(propertyName = "destinationType",
7       propertyValue = "javax.jms.Queue")
8   })
9   public class MessageBean {
10    @Resource private MessageDrivenContext mdc;
11
12    public MessageBean() { } // constructor
13
14    @PostConstruct obtainResources() { }
15
16    @PreDestroy releaseResources() { }
17
18    public void onMessage(Message inMessage) { }
19  }
```

Summary

The following topics were presented in this module:

- The properties and life cycle of message-driven beans
- Creating a JMS message-driven bean
- Lifecycle event handlers for a JMS message-driven bean

The following are some interesting topics covered in SL-351-EE6, Advanced EJB Development.

- Message Selectors
- Acknowledgement modes

Module 13

Web Services Model

Objectives

Upon completion of this module, you should be able to:

- Describe the role of web services
- Distinguish between the two major types of web services
- List the specifications used to make web services platform independent
- Describe the Java APIs used for XML processing and web services

Additional Resources

Additional resources – The following references provide additional information on the topics described in this module:

- "JSR 224: Java API for XML-Based Web Services (JAX-WS), Version 2.2", [http://jcp.org/en/jsr/detail?id=224], accessed 1 August 2009.
- "JSR 311: Java API for RESTful Web Services (JAX-RS), Version 1.1", [http://jcp.org/en/jsr/detail?id=311], accessed 1 August 2009.
- Eric Jendrock, Debbie Carson, Ian Evans, Devika Gollapudi, Kim Haase, Chinmayee Srivathsa. “The Java EE 6 Tutorial,” [http://java.sun.com/javaee/6/docs/tutorial/doc/], accessed 1 August 2009.

The Role of Web Services

The World Wide Web Consortium (W3C) defines a web service as “a software system designed to support interoperable machine-to-machine interaction over a network.” A web service exposes a remote *service* or executable procedure to a client application. Web services are designed to be platform independent; however, this platform independence is not without some cost. The cost of a web service’s platform independence is overhead, both in network and CPU usage.

Some key features of web services are:

- Platform Independent – No CPU, operating system, or programming language-specific data or functionally is required for web services.
- Designed to leverage existing technologies – Web services make extensive use of XML and HTTP technologies. Both of these technologies have large toolset and knowledge bases.
- Interoperable across disparate programming languages – Web services use a client-server model. It is possible to have a client written in one programming language communicating with a server written in a different language.

A client-server application with both its client and server written in the Java programming language that is not used with clients or servers written in other languages, has little need to function as a web service. Web services have been supported by the enterprise Java platform since J2EE 1.4 and even though web services are no longer a new technology there is still a certain amount of industry excitement about them. You may be required to make an application function as a web service without a technically compelling reason. Luckily, the Java EE 6 platform enables the easy creation or addition of web services to your applications.

Web Services as Remote Components

Web services provide a mechanism to remotely execute a business operation similar to a remote session EJB. Web services provide remote execution similar to:

- Common Object Request Broker Architecture (CORBA) – A specification managed by the Object Management Group (OMG). Provided bindings for the C programming language and later C++ followed by Java. Seen by some as overly complex. Used mainly in the 1990s. Beginning in 2000, web services started supplanting CORBA for some applications.
- Remote Method Invocation (RMI) – A Java technology remote object method invocation technology. The preferred method for basic distributed Java components. Included as part of the Java SE platform.
- Remote Procedure Call (RPC) – A procedural remote execution protocol popularized by Sun Microsystems. Not typically used in object-oriented software. Network File System (NFS) is an example of a RPC program.
- Distributed Component Object Model (DCOM) – The Microsoft equivalent to RMI, which worked only on Microsoft platforms. DCOM has been replaced by .Net Remoting and web services.

Web services differ from these technologies by leveraging the Hyper Text Transfer Protocol (HTTP) and Extensible Markup Language (XML), both of which are well supported by libraries in almost every programming language. Java libraries supporting web services are supplied in both the enterprise version of the Java platform and the standard edition platform.

Web Services Compared to Remote EJBs

The overall architecture of remote EJBs is not that different from web services, with the exception of the specific technologies used to make the web services platform independent:

- A registry for publishing and looking up web services. Registries implement the Universal Description, Discovery and Integration (UDDI) standard. If the client already knows the location of the desired service, you can bypass the registry.
- A transport protocol used to invoke operations, pass arguments, and receive return values. In remote EJBs this can be the Internet Inter-Orb Protocol (IIOP). In web services HTTP is typically used.
- A sequence of data transferred between client and server. Remote session beans use a serialized object while some web services use XML.

Currently most implementations of web services are stateless so they are best compared to stateless session beans.

Note – Sun offers a five day instructor-led training course dedicated to web services entitled *Developing Web Services Using Java Technology* (DWS-4050).

Web Service Standards

There are two prevailing styles of web services which are discussed in the following sections.

RESTful Web Services

Representational State Transfer (REST) is an architectural style. A web service that follows this architectural style is said to be RESTful. RESTful web services revolve around resources. A resource many be any *thing* such as a stock, customer, or share of stock. The state of a resource is captured and transferred between a client and server, hence the architecture is called Representational State Transfer.

Note – For more information on REST architecture see Roy Fielding's dissertation, *Architectural Styles and the Design of Network-based Software Architectures*, available at `http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`.

A resource is given a URI such as `http://localhost/stocks/ORCL`. HTTP methods are used to perform operations relating to that resource.

- HTTP GET - Retrieve a representation of the resource.
- HTTP POST - Add a new element to a collection such as adding a stock to the `http://localhost/stocks` resource.
- HTTP PUT - Create or update the resource.
- HTTP DELETE - Delete the resource.

The HTTP body content type may be any MIME type the developer wishes to use.

RESTful web services benefit from their simplicity and the wide adoption of HTTP.

SOAP Web Services

A web service that follows the Simple Object Access Protocol (SOAP) specification when exchanging XML formatted information is referred to as a SOAP web service. The Java EE Tutorial refers to these as ”big” web services.

SOAP web services can be more complex and require more plumbing but do have benefits. These types of services are have a machine readable API document called a WSDL file which prompts tool generated stubs for easy utilization in any platform.

SOAP web services typically involve standards and specifications not used in RESTful web services such as those prompted by the Web Services Interoperability Organization (WS-I).

Note – For more in-depth analysis comparing RESTful web services to SOAP web services see *RESTful Web Services vs. ”Big” Web Services: Making the Right Architectural Decision*, `http://www2008.org/papers/pdf/p805-pautassoA.pdf`.

Interoperability Requirements

Web services are designed to be platform and language neutral. To make web services effective, they must:

- Support clients regardless of platform or language – A web service does not depend on clients being coded in a certain programming language or executed on a specific operating system.
- Be able to be implemented in any language regardless of platform
  - A RESTful web service may not have a machine digestible API description. There are several standards being proposed to describe the API of a RESTful web service. Not having an industry-wide adopted standard to describe RESTful web services does not prevent them from being used. A well designed REST architecture ensures that a web service will be easy to understand to humans. The service may be documented in a human readable format.
  - A SOAP web service exposes all API information with a text file known as a Web Services Description Language (WSDL) file. With a WSDL file, you can develop a new client in any programming

language without access to any other server information. Most Java tools also support the creation of compatible replacement web services with nothing more than a WSDL file.

Two of the specifications used to make web services interoperable are HTTP and XML. HTTP is a transport protocol, typically used on top of Transmission Control Protocol/Internet Protocol (TCP/IP), that can be used to transfer information in a request-response oriented architecture. XML is a specification that is used to organize data in a machine-parsable text file. Web services use HTTP to send formatted requests and responses, possibly as XML messages.

Using HTTP as a transport protocol and XML as a data format does not provide a high level of interoperability. To be interoperable:

- The methods and headers used in the HTTP transport protocol must be standardized.
- The format of messages must be known. There are many available data interchange formats such as Comma Separated Values (CSV), JSON, and XML. These formats requires a document to follow a certain basic syntax by themselves doe not specify a complete structure.
  - RESTful web services may use any data format. Often some text based MIME type will be selected. If XML is used developers may created and public custom XML schema definitions.
  - SOAP web services use XML and the Simple Object Access Protocol (SOAP). SOAP is the specification that outlines a standard structure to the XML messages used in web services.

Note – The World Wide Web Consortium (W3C) publishes the HTTP, XML and SOAP specifications at `http://www.w3c.org`

SOAP Interoperability Standards

XML, HTTP, and SOAP are not the only specifications used to achieve interoperability between web services and their clients. Because many specifications are used in web services, specifications have been created to mandate the specification and version to be used. This is similar to how Java EE 6 mandates the use of specifications, such as EJB 3.1 and Servlet 3.0.

- XML – A W3C standard designed to store data in a format that is both readable to humans and easily parsed by machines.
- HTTP – A W3C standard for short lived data communication.
- UDDI – A publish and lookup standard sponsored by OASIS.
- SOAP – A W3C standard that enforces a more defined structure upon the XML formatted messages used in web services.

Code 13-1 Example Soap Request

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:ns1="http://sample/">
<soapenv:Body>
<ns1:getGreeting>
<arg0>Duke</arg0>
</ns1:getGreeting>
</soapenv:Body>
</soapenv:Envelope>
<?xml version="1.0" encoding="UTF-8"?>
```

Code 13-2 Example Soap Response

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:ns1="http://sample/">
<soapenv:Body>
<ns1:getGreetingResponse>
<return>Hello Duke</return>
</ns1:getGreetingResponse>
</soapenv:Body>
</soapenv:Envelope>
```

- WSDL – A proposed W3C standard that defines the functionality of a web service. Similar in purpose to the business interface of a Session EJB component, but formatted in XML.

Code 13-3 Example WSDL File

```
<?xml version="1.0" encoding="UTF-8"?><definitions
xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:tns="http://sample/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
targetNamespace="http://sample/" name="SayHelloService">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://sample/"
schemaLocation="http://10.1.1.1:8080/WebService/SayHelloService/__contain
er$publishing$subctx/WEB-INF/wsdl/SayHelloService_schema1.xsd"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"/>
    </xsd:schema>
  </types>
  <message name="getGreeting">
    <part name="parameters" element="tns:getGreeting"/>
  </message>

  <message name="getGreetingResponse">
    <part name="parameters" element="tns:getGreetingResponse"/>
  </message>
  <portType name="SayHello">
    <operation name="getGreeting">
      <input message="tns:getGreeting"/>
      <output message="tns:getGreetingResponse"/>
    </operation>
  </portType>

  <binding name="SayHelloPortBinding" type="tns:SayHello">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
style="document"/>
    <operation name="getGreeting">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>

      </output>
```

```
    </operation>
  </binding>
  <service name="SayHelloService">
    <port name="SayHelloPort" binding="tns:SayHelloPortBinding">
      <soap:address
location="http://10.1.1.1:8080/WebService/SayHelloService"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"/>
    </port>
  </service>
</definitions>
```

- Web Services Interoperability (WS-I) Basic Profile – A profile that mandates the use of certain specifications, such as SOAP and WSDL, and placing additional restrictions on them as needed, to provide interoperable web services.

Java APIs Related to XML and Web Services

The specifications in the following list can be used when providing or using a web service in the Java programming language. However, you may not have to use or even know the details of these specifications to use Java technology-based web services.

- The Java Document Object Model (JDOM or Java DOM) provides a object-oriented Java model of an XML document.
- The Java API for XML Processing (JAXP) is a Java technology abstraction of the actual XML processing implementation.
- The Java Architecture for XML Binding (JAXB) controls how to convert Java objects to XML schemas.
- The Java API for XML-based Remote Process Communications (JAX-RPC) is the original high-level web services specification for Java technology, replaced by the Java API for XML Web Services (JAX-WS). JAX-RPC is a proposed optional specification in Java EE 6.
- The Java API for XML Registries (JAXR) is a Java standard for using UDDI registries. JAXR is a candidate for pruning in future versions of the Java EE platform.
- The SOAP With Attachments API for Java (SAAJ) is a Java standard for creating, sending, receiving, and parsing SOAP messages. Used by JAX-WS.
- The Java API for RESTful Web Services (JAX-RS) is a high level Java web service API. JAX-RS implementations rely on HTTP as the communication protocol. Developers may use Java XML APIs is producing or consuming XML messages.
- The Java API for XML Web Services (JAX-WS) is a high level Java web service API. JAX-WS implementations make use of implementations of SAAJ, which in turn make use of JAXP and other Java XML specifications.

The JAX-RS API

The JAX-RS API is the highest level Java API for RESTful web services. JAX-RS:

- Was first included in the Java EE platform
- Requires no specific data interchange format
- Developers may use JAXB or any other Java XML API if they choose to produce or consume XML
- Provides only server APIs

Using JAX-RS on the server side does not require its use on the other side of the connection. If you are developing RESTful web services using Java technology, then you should use the JAX-RS API.

The JAX-WS API

The JAX-WS API is a high level Java API for SOAP web services. JAX-WS:

- Replaces JAX-RPC
- Requires little to no XML or WSDL knowledge for basic web services
- Uses JAXB to specify how Java technology and XML data types are mapped
- Provides both client and server APIs

Using JAX-WS on the client or server side does not require its use on the other side of the connection. If you are developing SOAP web service clients and servers using Java technology, then you should use the JAX-WS API.

Figure 13-1 Illustrates how JAX-WS technology manages communication between a web service and client.

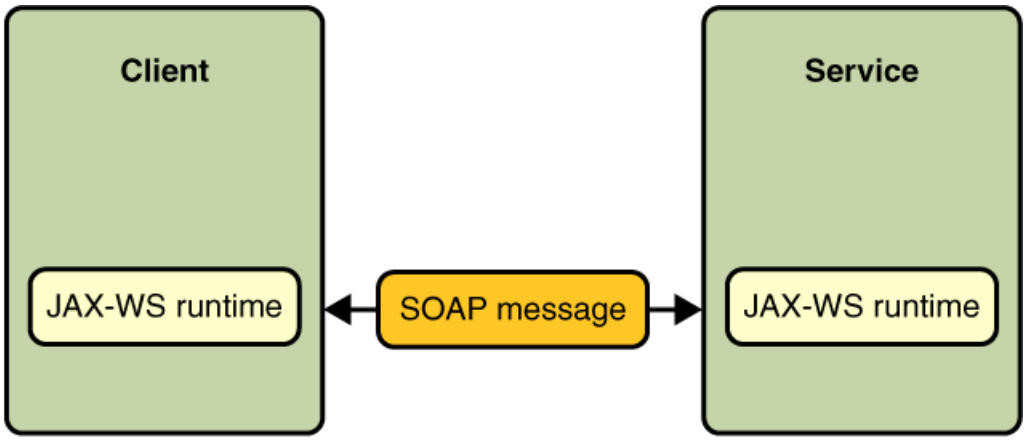


Figure 13-1 Communication Between a JAX-WS Web Service and Client

Web Service Benefits

Even though there is little use for creating web services that have both clients and services implemented only in the Java platform, there are still some reasons to consider using web services:

- The flexibility of supporting unknown future client platforms.
- Not being tied to a particular server platform. For example, if the server platform does not scale well, it can be replaced without affecting client applications.
- The use of HTTP as a communication technology allows the use of existing firewall rules in many cases.

Summary

The role of web services in enterprise applications is to provide a platform-neutral method for remote execution of code. To be platform-neutral, several specifications are used, such as SOAP and HTTP. Java technology provides APIs for XML processing and web services.

Module 14

Implementing Java EE Web Services with JAX-RS & JAX-WS

Objectives

Upon completion of this module, you should be able to:

- Describe endpoints supported by the Java EE 6 platform
- Describe the requirements of JAX-RS POJO endpoints
- Describe the requirements of JAX-WS POJO endpoints
- Develop web service clients

Additional Resources

Additional resources – The following references provide additional information on the topics described in this module:

- "JSR 224: Java API for XML-Based Web Services (JAX-WS), Version 2.2", [http://jcp.org/en/jsr/detail?id=224], accessed 1 August 2009.
- "JSR 311: Java API for RESTful Web Services (JAX-RS), Version 1.1", [http://jcp.org/en/jsr/detail?id=311], accessed 1 August 2009.
- Eric Jendrock, Debbie Carson, Ian Evans, Devika Gollapudi, Kim Haase, Chinmayee Srivathsa. “The Java EE 6 Tutorial,” [http://java.sun.com/javaee/6/docs/tutorial/doc/], accessed 1 August 2009.
- “JSR 109: Implementing Enterprise Web Services", [http://jcp.org/en/jsr/detail?id=109], accessed 1 August 2009.
- “JSR 181: Web Services Metadata for the Java Platform, Version 2.1", [http://jcp.org/en/jsr/detail?id=181], accessed 1 August 2009.

Web Service Endpoints Supported by the Java EE 6 Platform

A web service endpoint is the remotely executable component that exists on a server and is executed as the result of receiving a web service request from a web service client. JAX-RS and JAX-WS, which art part of Java EE 6, outline two types of components that can function as an endpoint:

- POJO components – Both JAX-RS and JAX-WS provide annotations that are used to configure a plain Java class to function as a web service endpoint. In JAX-RS an endpoint is sometimes known as a resource class. In JAX-WS this endpoint is sometimes known as a servlet or web endpoint.
- Session bean components – Stateless and Singleton session beans components can function as a web service endpoint. Stateful session beans can not function as endpoints.

The choice between JAX-RS and JAX-WS will largely depend on non-functional requirements.

JAX-RS Web Endpoints

The Java API for RESTful Web Services (JAX-RS) is a specification that, along with an implementation, supports the execution of remote methods using the HTTP protocol for message transport. The messaging format used is selected by the developer. Sun provides an reference implementation (RI) of JAX-RS named Jersey.

When a RESTful web service client wants to execute a remote method, it delivers a message to a server using the HTTP protocol methods. It is the job of the JAX-RS implementation to convert the HTTP request to a Java method call by using the request URI and HTTP method type. You must supply the methods called by JAX-RS and these methods typically exist in an *endpoint* component. In JAX-RS a endpoint component is also known as a resource class.

A JAX-RS endpoint is a POJO class also known as a resource class. When deployed to a web container, the JAX-RS framework provides a servlet implementation used to handle HTTP requests. A JAX-RS web endpoint:

- Is an annotated Java class created to provide web service functionality.
- Is instantiated per request
- Does not require an EJB container. This is the main benefit of this type of endpoint. Servlet and JSP containers, such as Apache's Tomcat can be set up to support this type of web service.

Implementing a JAX-RS Web Endpoint

A JAX-RS web endpoint has the following requirements:

- Has a `@javax.ws.rs.Path` class annotation
- Have business methods that are public and are annotated with a request method designator (e.g. `@GET`)
- Uses the `@Produces` and/or `@Consumes` annotations at the class or method level
- Cannot be an abstract
- Usually has a no-arg constructor

Code 14-1 Simple JAX-RS Web Component Endpoint Example

```
package com.example;

import javax.ws.rs.*;

@Produces("text/plain")
@Consumes("text/plain")
@Path("greeting")
public class GreetingResource {

    private static String message = "Howdy";

    public GreetingResource() {}

    @GET
    public String getGreetingText() {
        return message;
    }

    @PUT
    public void putText(String content) {
        message = content;
    }

}
```


Note – The example above does not address thread safety.

JAX-RS Servlet Endpoint Configuration

You must update the `web.xml` in your web archive (war) to provide a usable URL for the web service. The `<load-on-startup>` element should be specified so that the web server loads this class when starting up. When classes are loaded that contain a `@Path` annotation, the JAX-RS provided servlet handles all HTTP requests for the configured URL.

Code 14-2 JAX-RS Web Component Endpoint `web.xml` File

```
<servlet>
   <servlet-name>ServletAdaptor</servlet-name>
   <servlet-class>
   com.sun.jersey.spi.container.servlet.ServletContainer
   </servlet-class>
   <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
   <servlet-name>ServletAdaptor</servlet-name>
   <url-pattern>/resources/*</url-pattern>
</servlet-mapping>
```

JAX-RS URI Parameters

The URI or Path used to invoke a JAX-RS web service can contain embedded parameters.

```
@Path("orders/{order_id}")
public class OrderResource {

   @GET
   Order getOrder(@PathParam("order_id") String id) {
      //...
   }
}
```

JAX-WS Web Endpoints

The Java API for XML Web Services (JAX-WS) is a specification that, along with an implementation, supports the execution of remote methods using an XML-based messaging schema. The protocol used for message transport is HTTP and the XML messaging format used is SOAP. Sun provides a Reference Implementation (RI) for application servers to provide JAX-WS support.

When a "big" web service client wants to execute a remote method, it delivers a SOAP message to a server that, in this case, uses JAX-WS. It is the job of the JAX-WS implementation to convert the SOAP message to method calls and arguments. You must supply the methods called by JAX-WS and these methods only exist in an *endpoint* component.

The JAX-WS framework provides a servlet implementation used to handle HTTP requests and process SOAP messages. A JAX-WS web endpoint:

- Is an annotated Java class created to provide web service functionality.
- Is multi-threaded
- Does not require an EJB container. This is the main benefit of this type of endpoint. Servlet and JSP containers, such as Apache's Tomcat can be set up to support this type of web service.

Implementing a JAX-WS Web Endpoint

A JAX-WS web endpoint has the following requirements:

- Has a `@javax.jws.WebService` class annotation
- Must have business methods that are public and not final or static
- Contains exposed web service methods that are annotated with `@javax.jws.WebMethod`
- Cannot be an abstract or final class
- Requires a default no-arg constructor

Code 14-3 Simple JAX-WS Web Component Endpoint Example

```
package com.example;

import javax.jws.*;

@WebService
public class SayHello {

   @WebMethod
   public String getGreeting(String name)  {
      return "Hello " + name;
   }
}
```

JAX-WS Servlet Endpoint Configuration

You must update the `web.xml` in your web archive (war) to provide a usable URL for the web service. The `<load-on-startup>` element should be specified so that the web server loads this class when starting up. When classes are loaded that contain a `@WebService` annotation, the JAX-WS provided servlet handles all HTTP requests for the configured URL.

Code 14-4 JAX-WS Web Component Endpoint `web.xml` File

```
<servlet>
    <servlet-name>hello</servlet-name>
    <servlet-class>example.SayHello</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>hello</servlet-name>
    <url-pattern>/sayhello</url-pattern>
</servlet-mapping>
```

JAX-WS Endpoint Life Cycle

JAX-WS endpoints, both web and EJB, can have optional life-cycle methods that are automatically called if present. Any method can be used as a life-cycle method with the correct annotation:

- `@PostConstruct` – Called by the container before the implementing class begins responding to web service clients.
- `@PreDestroy` – Called by the container before the endpoint is removed from operation.

A web component endpoint follows the typical servlet threading model, which means that typically there is one instance that is executed concurrently for each client. JAX-WS implementations are free to leverage pools of bean instances to handle a request in a fashion similar to stateless session EJB components.

JAX-WS Allowed Data Types

Unlike the JAX-RPC, the JAX-WS does not contain a Java-to-XML binding specification for data types. The Java Architecture for XML Binding (JAXB) specification is designated by JAX-WS as the way in which Java data is converted back and forth to XML.

Basic Java types, such as Strings are supported automatically but to return or pass complex object instances, some JAXB programming is required.

While outside the scope of this course you should know that JAXB is not only used with JAX-WS. You can use JAXB any time you need to convert a Java object to an XML document. JAXB also supports generating classes from XML schemas and XML schemas from Java classes.

Code 14-5 Basic JAXB Annotated Class That Could Be Returned From a JAX-WS Method

```
import javax.xml.bind.annotation.XmlType;

@XmlType
public class Person {
    private String name;
    private int age;

    public Person() { }
    public Person(String name, int age) {
        this.setName(name);
        this.setAge(age);
    }

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
}
```

Code 14-6 SOAP Response of a JAXB Annotated Object

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope
/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:ns1="http://server/">
<soapenv:Body>
<ns1:getPersonResponse>
<return>
<age>40</age>
<name>Bob</name>
</return>
</ns1:getPersonResponse>
</soapenv:Body>
</soapenv:Envelope>
```

Web Service Clients

Java web service clients are programmed differently for RESTful and SOAP web services. The following sections cover creating RESTful and SOAP web service clients in Java.

RESTful Web Service Clients

There is no official JAX-RS client API at this time. Since RESTful web services are much simpler than SOAP web services it is possible to use `java.net.HttpURLConnection` as a means to develop a Java RESTful web service client.

The reference implementation of JAX-RS does include a RESTful web service client API. For details and documentation visit `https://jersey.dev.java.net/`.

A RESTful Client Example

```
package com.example;

import java.io.OutputStreamWriter;
import java.net.HttpURLConnection;
import java.net.URL;

public class Client {

    public static void main(String[] args) throws Exception {
        URL url =
          new URL("http://localhost:8080/JAXRSDemo/resources/greeting");
        HttpURLConnection conn = (HttpURLConnection)url.openConnection();
        conn.setDoOutput(true);
        conn.setRequestMethod("PUT");
        OutputStreamWriter out =
          new OutputStreamWriter(conn.getOutputStream());
        out.write("Hello");
        out.close();

        System.out.println("Response: " + conn.getResponseCode());
        conn.disconnect();
    }
}
```

Non-Java RESTful Web Service Clients

JAX-RS web services are platform neutral and can be called by any client. To develop clients in other platforms a developer will need to:

- Have access to some type of documentation describing the web service API
- Be able to produce and consume data in the format utilized by the web service

Note – Several technologies exist for RESTful web service description formats including WSDL 2.0 and WADL but none have achieved a large market share.

JAX-WS Web Service Clients

The key to creating a SOAP web service client, in any language, is having a copy of the web service's Web Services Description Language (WSDL) file. The WSDL describes the operations, arguments, and return values used in a web service.

You can create a JAX-WS web service with either a WSDL file or a Java class annotated with `@WebService`. The reference implementation provides two tools to generate either the Java classes or WSDL file: `wsgen` and `wsimport`. You can run these tools manually; however, the WSDL file for a JAX-WS web service is generated automatically when the web service is deployed, so it is unnecessary.

Although not part of the specification, to retrieve the generated WSDL file from the GlassFish Application Server, you can use your browser by requesting a URL similar to:

```
http://localhost:8080/WSApp-war/SayHelloService?WSDL
```

Developing JAX-WS Clients

To access a web service from a JAX-WS client, you need:

- The WSDL file – Any web service client is unaware of how the service is coded. The WSDL file provides the information to create client-side interfaces for web services.
- A Proxy object to handle the creation of SOAP messages and HTTP communication. This proxy type class is known as a `Port` class in JAX-WS.
- A generated `Port` class or alternative source code and any other required artifacts for the web service.

The JAX-WS reference implementation bundled with the GlassFish Application Server provides a command called `wsimport` to create all the necessary client code. You can execute the `wsimport` tool by an IDE build process, typically as an Ant task.

A JAX-WS Client Example

In the following example, the `Service` class is used to instantiate a `Port` or proxy. The `Port` class handles all SOAP message creation and transmission.

Code 14-7 JAX-WS Client Example

```
public class WSTest {

public WSTest() { }

   public static void main(String[] args) {
   SayHelloService service = new SayHelloService();
   SayHello port = service.getSayHelloPort();
   System.out.println(port.sayHello("Duke"));
}
}
```

Non-Java SOAP Web Service Clients

JAX-WS web services are platform neutral and can be called by any client. In fact, a JAX-WS client never even knows whether it is using a JAX-WS service. If you must support clients developed with other platforms or languages, then that client must:

- Have access to the WSDL file
- Support the correct version of the SOAP specification.

For maximum compatibility, the client should support the WS-I Basic Profile 1.1.

Note – You can find additional information about WS-I and achieving interoperable web services at the Web Services Interoperability Organization's web site `http://www.ws-i.org/`

Summary

The Java EE 6 platform supports two different types of web services. JAX-RS enables developers to create RESTful web services. JAX-WS enables developers to create SOAP web services and clients.

Module 15

Implementing a Security Policy

Objectives

Upon completion of this module, you should be able to:

- Exploit container-managed security
- Define user roles and responsibilities
- Create a role-based security policy
- Use the security API
- Configure authentication in the web tier

Additional Resources

Additional resources – The following references provide additional information on the topics described in this module:

- Eric Jendrock, Debbie Carson, Ian Evans, Devika Gollapudi, Kim Haase, Chinmayee Srivathsa. “The Java EE 6 Tutorial,” [`http://java.sun.com/javaee/6/docs/tutorial/doc/`], accessed 1 August 2009.
- "Java Platform, Enterprise Edition 6 (Java EE 6) Specification", [`http://jcp.org/en/jsr/detail?id=316`], accessed 1 August 2009.
- "JSR 318: Enterprise JavaBeans, Version 3.1", [`http://www.jcp.org/en/jsr/detail?id=318`], accessed 1 August 2009.
- "JSR 315: Java Servlet Specification, Version 3.0", [`http://www.jcp.org/en/jsr/detail?id=315`], accessed 1 August 2009.

Exploiting Container-Managed Security

The Java EE specification defines an *end-to-end*, *container-managed*, *role-based*, *vendor-neutral* security model. In this model, your main task is to define roles and create the *declarative* security policy. Declarative in this context means that the security policy is described in the deployment descriptor, and not implemented in code. While the policy describes what operations are permitted to various groups of users, it does not describe how this restriction should be implemented.

Applications are not required to make use of the security model. There is nothing in the Java EE specification that prohibits you from implementing all security features in code. However, the Java EE security model is powerful, portable, easy to use, and suitable for a great many applications. If you decide to use your own security implementation, keep in mind that the Java EE security model is all-or-nothing. Security systems written by you cannot interact with the container's security mechanisms. You cannot use some parts of the security model and not use other parts of the model.

For example, it is recognized that some access control requirements are too fine-grained to be implemented declaratively. Therefore, the Java EE specification defines a security API that allows such operations to be carried out in code. However, the API calls only produce meaningful results when they are used in the context of container-managed authentication. If you implement your own authentication procedure, you cannot expect the application server's access control infrastructure to understand it. In other words, if you take on responsibility for security, you must do everything yourself.

Security Concepts

The following basic concepts are relevant to understanding the Java EE security model:

- Authentication – The process of establishing that a user or client really is who he or she claims to be.

  Authentication can be as simple as prompting for a user name and password. More complex systems might make use of a public-key infrastructure to manage the certificate exchange. In the Java EE model, authentication is carried out by the containers and not by the components.

- Authorization – The process of establishing whether a user is allowed to carry out a requested operation.

  Normally, it is necessary to authenticate a user before authorization can be checked, although an unauthenticated user might be allowed to carry out certain operations. The Java EE model primarily uses declarative authorization.

- Confidentiality – The protection of data from unauthorized viewing during transmission.

  The normal approach is to use encryption that is usually based on public-key techniques.

- Integrity – The process of ensuring that the data that is received is the same as the data that was sent.

  This process is normally achieved by *signing* the data. That is, the security model encrypts a checksum of the data and adds the encrypted checksum to the message to ensure that the message has not been modified during transmission.

In practice, confidentiality and integrity are dealt with at the transport level using a low-level, public-key protocol, such as secure sockets layer (SSL). SSL is now formally known as transport layer security (TLS). The use of TLS for component interactions is frequently standard in Java EE applications. Because the configuration of the transport is part of the administration of the application server and is not normally visible to the application developer, this module does not present transport-level security.

End-to-End Security

The Java EE security model is described as *end-to-end* because after a user's identity has been established on behalf of some component, that identity can be propagated to other components. Figure 15-1 shows an example of an end-to-end security model.

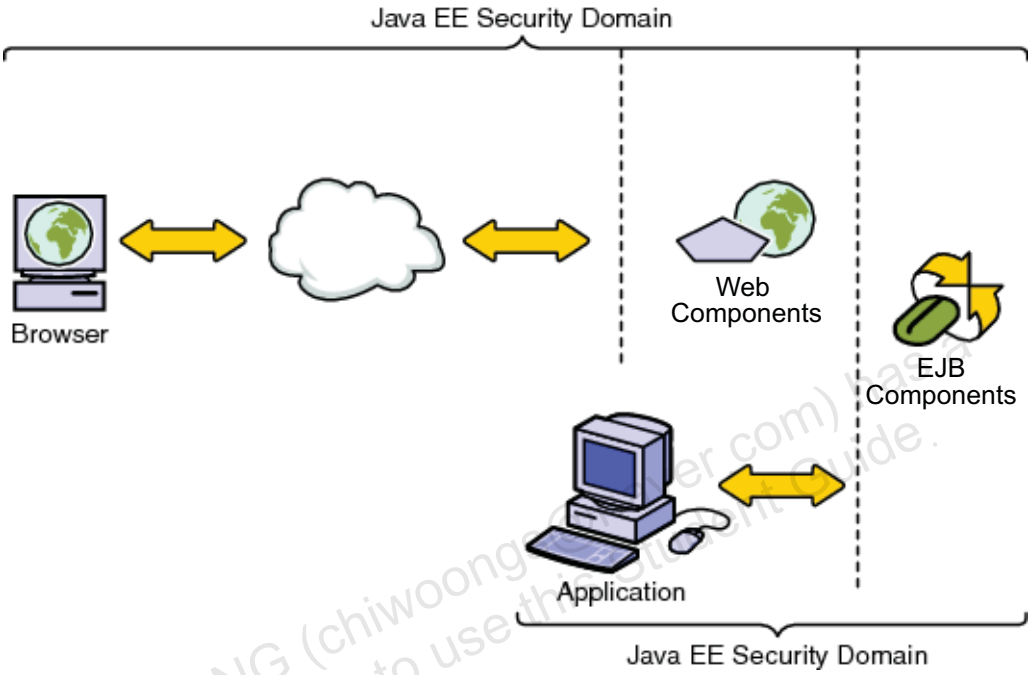


Figure 15-1 End-to-End Security Model

The following examples illustrate what end-to-end security means in practice:

- If a user is authenticated at the web tier, then the user's identity is made available to an EJB component when servlets in the web tier make method calls on an EJB component.
- If the EJB component has an access control restriction in place that allows a certain user to execute a certain method, then it is of no importance that the user was first authenticated to execute a servlet. The user is still subject to the EJB component's access control restrictions.
- If a standalone Java application makes method calls on EJB components, and is hosted by an application client container, then the client container can authenticate the user and pass the user's identity on to the EJB component.

In addition, if one component makes access control decisions based on whether the user is a member of a particular security role, you can be sure that other components that use the same roles will make consistent decisions, because role membership information is propagated between containers.

This membership propagation between containers makes development more straightforward, because you do not have to be concerned about how security information is passed from one part of a complex system to another.

Container-Managed Security

The application server is responsible for the implementation of both authentication and authorization, which leaves you to establish the policy that will be enforced. This is possible because components in the Java EE model never interact directly, but only through their containers. Therefore, it is the containers that enforce the security policy.

Figure 15-2 shows how a security policy is enforced by the web containers and the EJB containers. Typically, each time a web operation is initiated by a browser that issues a request, the web container checks the user's security privileges against the policy for the requested operation. If the operation is not allowed, then the application code is never invoked. Similarly, with EJB components, whenever a method call is made through the home object or component object, the caller's security privileges are checked against the policy for that method. If the method call is not allowed, the call is simply not made.

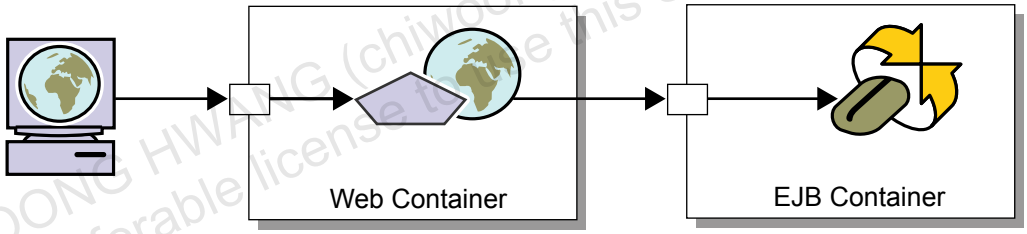


Figure 15-2 Security Policy Enforced by Web Containers and EJB Containers

Container-Managed Authentication

Authentication might be required in the web tier, the EJB tier, or in a standalone client. Remember that because the Java EE security model is end-to-end, authentication only happens once per session and at one point. After authentication is verified, the identity is propagated between components. Remember also that you cannot take over the authentication process without losing the ability to use other features of the Java EE security model.

Authentication in the Web Tier

If the users's identity is not known when a request is made for a web component, the web container can bring about an authentication sequence. Figure 15-3 shows the authentication process in the web tier.

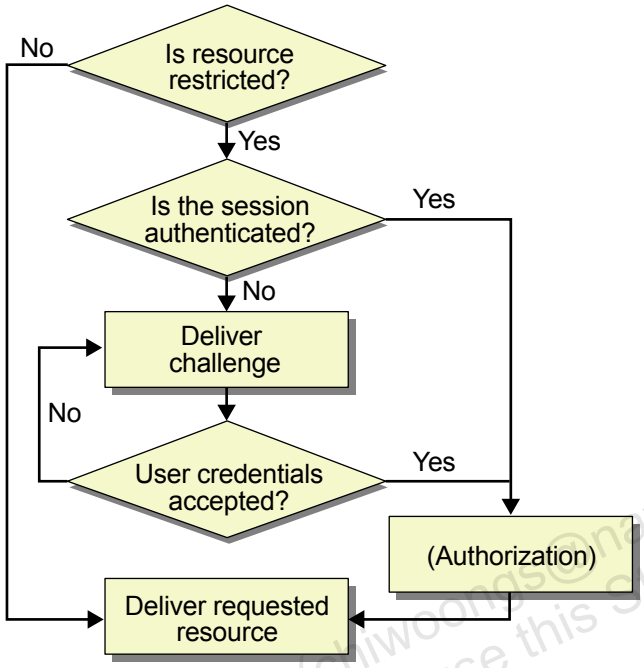


Figure 15-3 Authentication in the Web Tier

Again, you do not need to coordinate this process because it is entirely under the control of the containers. Web containers typically use *lazy authentication*. Lazy authentication means that the user is only asked to authenticate if the requested resource, such as a servlet or JSP component, is actually subject to a security restriction.

Therefore, authentication completes the following basic sequence:

- If the resource is not restricted, the resource is delivered without challenge.
- If the resource is restricted, the container gets the session ID from the request and checks in the session store to determine whether the user is already authenticated.
- If the user is authenticated, then there is no need to authenticate again.

  The container checks whether the user has permission to request the specified resource. This check is shown as (Authorization) in Figure 15-3 on page 15-7 because it is more involved than can be shown in the figure.

- If the user is not authenticated, then instead of delivering the requested resource, the container delivers an authentication challenge. The challenge very often consists of a request to the user to supply a user name and password.
- If the user name and password are acceptable, then the container moves on to the authorization step.

Web-Tier Authentication Challenge Methods

The Java EE specification defines three types of authentication that must be supported in the web tier.

- HTTP Basic – The web browser prompts the user for a user name and password, and supplies this information in the request header.

  HTTP basic authentication is the simplest method to use, and the least configurable. The appearance of the challenge is determined by the browser and not by the developer or the application server.
- Client Certificate – The client presents the user's digital certificate in response to a challenge from the server.

  Client certificate authentication is not widely used when the client is a web browser because end users tend to have certificates that the server is not prepared to trust. However client certificate authentication is important in a business-to-business context, where the client might be another enterprise application.
- Form-based – The developer controls the *look and feel* of the authentication process by supplying HTML forms.

  When authentication is required, the container provides your HTML instead of the requested resource. The actual authentication is completed by the container.
- *Programmatic* – Technically this is not a challenge method. Java EE 6 includes programmatic servlet security which adds the ability to force a challenge in-code or to collect authentication information in a custom way.

Figure 15-4 shows a basic authentication dialog box from a Netscape Navigator™ browser.

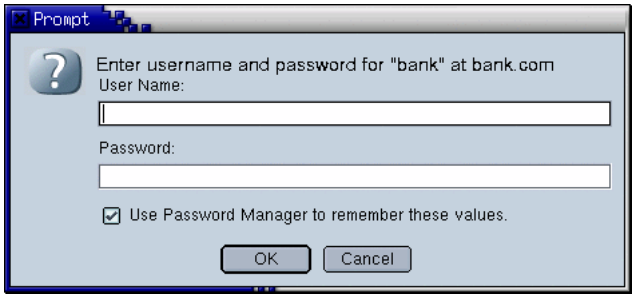


Figure 15-4 Basic Authentication Dialog Box

It is a common misconception that basic authentication is *less secure* than the other types of authentication. In fact, both basic and form-based methods of authentication supply the credentials in plaintext, unless a secure channel is established first. Consequently, basic authentication is no less secure than the other types in terms of cryptography.

The real problem with basic authentication is that after the user has logged in, it is difficult to log out. The reason for this problem is that after the browser has collected the user name and password, the browser can supply this information whenever the server asks for them. So the removal of the user's authenticated status at the application server only has the effect of making the browser resupply the credentials and log the user right back in again. There are solutions to this problem, but they are not entirely portable. Form-based authentication is usually the preferred method when the client is a web browser.

Authentication From the Web Tier to the EJB Tier

Do not attempt to supply authentication credentials to the EJB tier in code. A client of an EJB component must pass a verifiable security context with each method call that allows the user to be identified. When the client is a web component in the same application as the EJB component, it is the web container's responsibility to pass the user credentials to the EJB tier.

For example, there is never a good reason to pass the identity of the current user as an argument to a method of an EJB component. If the server has been configured correctly, the EJB component is always able to find the identity of the current user from the security context that is passed by the web container. When the client of the EJB component is *not* a web component, additional steps are usually required for authentication.

Authentication of Non-Web Clients

Authentication by the EJB components of clients other than components in the web tier is a large and complex subject area, and one that this course can address only at a high level. Web browser users are easy to authenticate because the web server can present its own authentication challenge to the user by way of the browser. No such procedure is defined for EJB components. When a client makes a method call on an EJB component, the method call will either succeed or it will fail. If the method call fails because the user is not identifiable, this failure cannot be rectified in the EJB tier.

Consequently, the client of an EJB component must pass the user's identity as part of the RMI protocol. Until recently, there was no standard method for doing this, so EJB servers and web servers from different vendors were not always interoperable.

If the client is an application in the Java programming language, then the use of an application client container simplifies matters. When the client container is generated, it contains vendor-specific code that knows how to communicate with the application server's security infrastructure. When it is necessary to authenticate a user, the client container prompts the user for credentials. Alternatively, the client container might accept credentials that are offered on the command line. The application is not required to complete authentication in either case. However, some products allow you to customize the look and feel of the authentication challenge by supplying a class that provides the user interface.

When the client is not implemented in the Java programming language, or when the use of a client container is not practical, then the client can use Internet Inter-ORB Protocol (IIOP) API calls directly to supply the security context. It has been possible to use IIOP API calls portably since the introduction of the Common Object Request Broker Architecture (CORBA) secure interoperability, version 2 (CSIv2) protocol. The portable use of IIOP API calls must now be supported by all application servers that support the Java EE platform 1.3 or higher. This support of IIOP API calls makes it possible for an application that is written in a software language, such as C++, to make method calls on EJB components and to authenticate itself by supplying a digital certificate. However, these procedures are beyond the scope of this course.

Interaction With the Security Infrastructure

When the user has supplied credentials to the application server, the server must check these credentials against some form of the security database. The security database might be part of the application server itself, or it might be an external server.

For example, it is common for large enterprises to maintain their users' security credentials in a dedicated directory server. This whole process is beyond the control of the application developer. However, the Java EE 6 specification mandates support for the Java Authentication and Authorization Service (JAAS) and Java Authorization Service Provider Contract for Containers (JACC) APIs. With these APIs, you can install new modules in the server to extend the *server's* authentication capability. This makes it possible to provide support for security infrastructures that are not handled out-of-the-box by the application server *without* having to complete authentication in the application itself.

User Roles and Responsibilities

The Java EE security model is *role-based*. A *role* is an abstraction of a set of user authorization privileges. Users in the same role have the same capabilities. The emphasis on roles allows the Java EE security model to be platform-independent.

Roles, Actors, and Use Cases

If you are familiar with UML modeling, you might find it helpful to think of a role as an *actor* in a use-case model. An actor is not a user. Rather, an actor is a collection of users with similar responsibilities. Each actor interacts with one or more use cases. A use case is a self-contained unit of capabilities within the application. It follows that the actor must have some access rights over that use case.

In Figure 15-5, the Customer role groups the rights and responsibilities that are shared by customers of the application, which includes the right to see the contents of their bank accounts. The Manager role groups the rights of managers, which includes the right to create new bank accounts, as well as to view existing bank accounts.

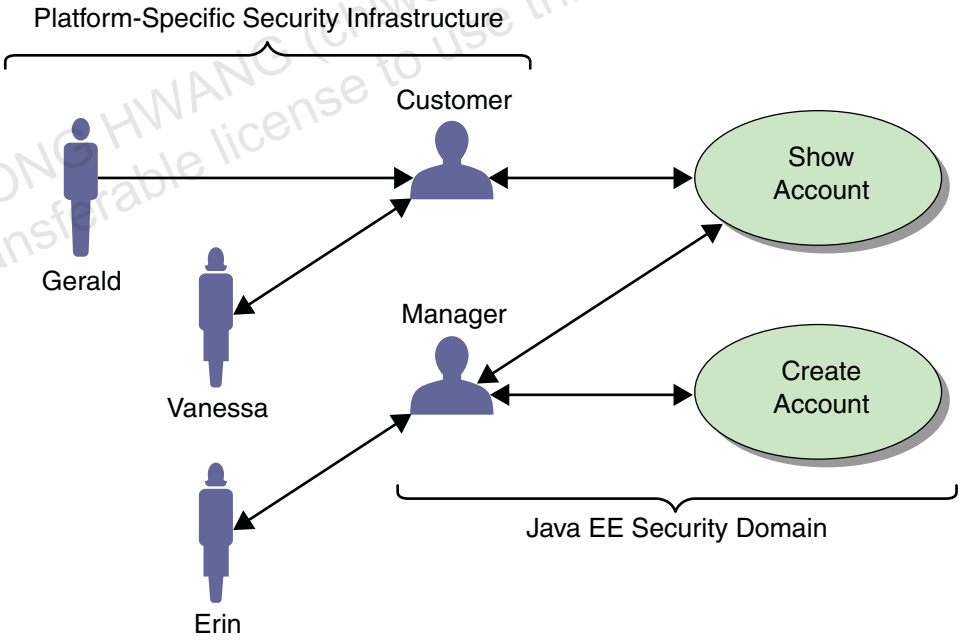


Figure 15-5 Role-Based Java EE Security Model

Unlike actors in UML, the role structure of the Java EE security model is *flat*, not hierarchical. That is, you cannot say that one role is a sub-type of another. The flat role structure is not a limitation in practice, because individual users can, and often do, occupy more than one role.

There is some correspondence between a role and a *group* in many practical security infrastructures, but the mapping of real users or groups to roles is platform-specific and not part of the Java EE model.

End-to-End Roles

Roles are defined at the application level, not at the level of components or tiers. While an application can have different roles in the web tier and the EJB tier, in practice, this is rarely useful. More commonly, roles are end-to-end. That is, roles are defined for the whole application and then applied in the web tier and EJB tier. This makes it easier to ensure a consistent security policy across the application.

Creating a Role-Based Security Policy

There are four major steps that are required to create an end-to-end, role-based security policy:

1. Clearly define the roles.
2. Assign roles to URL patterns in the web tier.

   Edit the `web.xml` file or use security annotations in Java EE 6.
3. Assign roles to methods in the EJB tier.

   Edit the `ejb-jar.xml` file or use the `@RolesAllowed` annotation on the EJB methods in Java EE 5 or 6.
4. Expose the roles that you have used.

   Edit the `application.xml` file or use the `@DeclareRoles` annotation in EJB components and Servlets in Java EE 5 or 6.

The determination of the roles requires knowledge of the application's overall structure and the responsibilities of its users. This determination is part of the design process and is not a development exercise. In practice, role assignments often come from use-case modeling, or simply from a knowledge of the different ways in which the users interact with the system.

Role Mapping

Security roles in a Java EE application must be mapped to either principals (user accounts) or groups. Role mapping information is placed in a vendor specific deployment descriptor file.

Code 15-1 An example `WEB-INF/sun-web.xml` file.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Application
Server 9.0 Servlet 2.5//EN"
"http://www.sun.com/software/appserver/dtds/sun-web-app_2_5-0.dtd">
<sun-web-app error-url="">
    <security-role-mapping>
        <role-name>user</role-name>
        <group-name>customers</group-name>
    </security-role-mapping>
</sun-web-app>
```

Role-Based Authorization in the Web Tier

Figure 15-6 shows the procedure that the web container uses to check the authorization of each request.

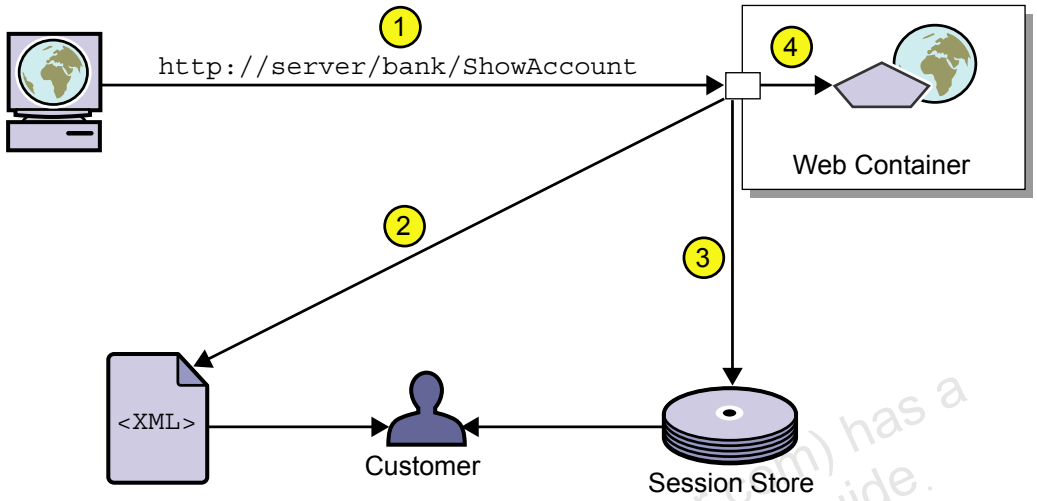


Figure 15-6 Role-Based Authorization in the Web Tier

The authorization procedure in the web tier involves the following steps:

1. The browser makes a request for a specific URL.
2. The URL of the request is matched against the patterns in the deployment descriptor that are subject to security restrictions and to security annotations placed in servlet classes.
3. If the URL is restricted, the server obtains the user's role assignments from the session store and the underlying security infrastructure.
4. The server checks the user's allocated roles against the roles allowed for the resource. If any of the user's roles are in the allowed set, the request is allowed, and the web component is invoked.

Web Tier Security Annotations

The following example shows an example of using security annotations to restrict access to a servlet component in Java EE 6. Java EE 5 and previous require the use of the web.xml deployment descriptor to configure security constraints.

Code 15-2 A Java EE 6 servlet with security annotations

```
package com.example;

import java.io.IOException;
import javax.annotation.security.*;
import javax.ejb.EJB;
import javax.servlet.*;
import javax.servlet.annotation.HttpConstraint;
import javax.servlet.annotation.HttpMethodConstraint;
import javax.servlet.annotation.ServletSecurity;
import javax.servlet.annotation.ServletSecurity.EmptyRoleSemantic;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.*;

@DeclareRoles({"admin", "user"})

// all HTTP methods require "admin" or "user"
// @ServletSecurity(@HttpConstraint(rolesAllowed = {"admin", "user"}))

// allow all HTTP methods except POST
// @ServletSecurity(httpMethodConstraints =
@HttpMethodConstraint(value="POST", emptyRoleSemantic =
EmptyRoleSemantic.DENY))

// GET requires "admin", other method are open
// @ServletSecurity(httpMethodConstraints = @HttpMethodConstraint(value =
"GET", rolesAllowed = "admin"))

// everything requires "admin" except GET
// @ServletSecurity(value = @HttpConstraint(rolesAllowed = "admin"),
httpMethodConstraints = @HttpMethodConstraint("GET"))

@WebServlet(name="SecurityServlet", urlPatterns={"/SecurityServlet"})
public class SecurityServlet extends HttpServlet {}
```

Role-Based Authorization in the EJB Tier

Figure 15-7 shows the procedure that the EJB container uses to check the authorization of each request.

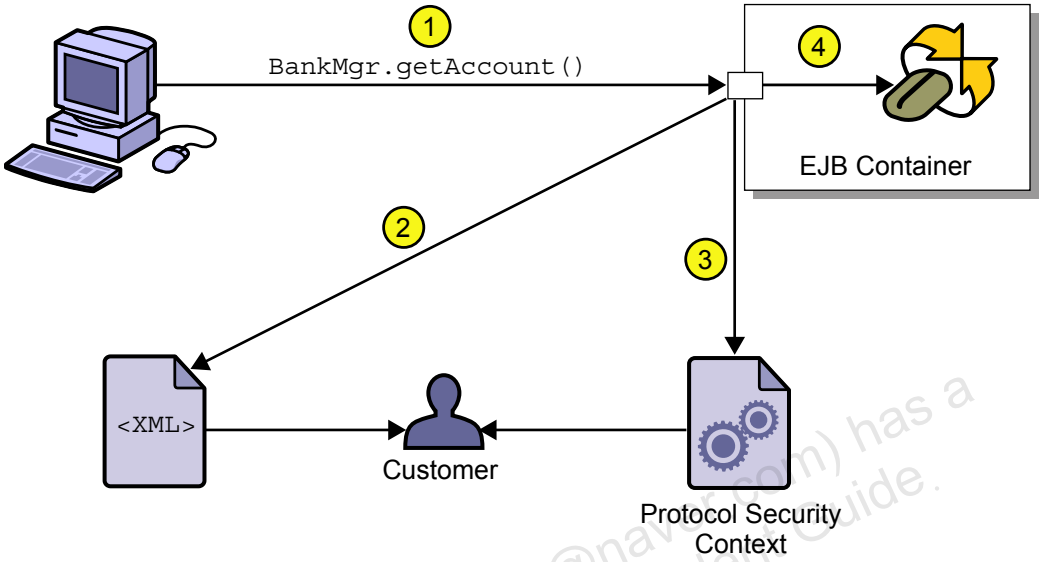


Figure 15-7 Role-Based Authorization in the EJB Tier

The authorization procedure in the EJB tier involves the following steps:

1. The client makes a specific method call on the EJB component.
2. The method call information is checked to determine whether this method has security restrictions assigned to it in the deployment descriptor or with annotations.
3. If the method is restricted, the server obtains the user's role assignments from the RMI protocol and the underlying security infrastructure.
4. The server checks the user's allocated roles against the roles that are allowed for the restricted method. If any of the user's roles are in the allowed set, the method is invoked. Otherwise the EJB container throws an exception.

EJB Tier Security Annotations

Java EE 5 and 6 allow security constraints to be placed on EJB components using annotations instead of using deployment descriptors.

Code 15-3 A Java EE 5/6 EJB component using security annotations

```
package com.example;

import javax.annotation.Resource;
import javax.annotation.security.DeclareRoles;
import javax.annotation.security.RolesAllowed;
import javax.ejb.SessionContext;
import javax.ejb.Stateless;

@Stateless
@DeclareRoles({"admin", "user"})
public class SecurityBean {

    @Resource SessionContext ctx;

    @RolesAllowed(value = {"admin", "user"})
    public String getText() {
        String name = ctx.getCallerPrincipal().getName();
        return name + ", your admin status is: " +
          ctx.isCallerInRole("admin");
    }
}
```

Using the Security API

The security API is only available to applications that use container-managed authentication. The API provides two basic facilities that you can use to determine the following information:

- The identity of the current user or client
- The user's role allocations

There are essentially only four method calls in the security API. The web tier uses the `getUserPrincipal` and `isUserInRole` methods. The EJB tier uses the `getCallerPrincipal` and `isCallerInRole` methods.

Web-Tier Security API

The web container makes the security context of the current user available with the `HttpServletRequest` object that is passed to the `service` method.

The following snippet shows the use of the `getUserPrincipal` method to determine the user or caller identity:

```
String user = request.getUserPrincipal().getName();
String message =
    "Welcome to the on-line bank, " + user;
```


The `getUserPrincipal` method returns a `Principal` object, which encapsulates the user identity. The `Principal.getName` method renders a textual form of that identity, which will be the user name if authentication is HTTP basic or form-based.

The following code snippet shows the use of the `isUserInRole` method to determine whether the user has been assigned to the manager role. A servlet might complete this method to construct a display that was appropriate for a particular user.

```
if (request.isUserInRole("manager")) {
  //... show manager menu
}
```

The literal `manager` that is used in the call to `isUserInRole` is not a role name, but it is a *role reference*. Role references are mapped onto real role names at deployment time. This process frees you from needing to know the final role names that will be assigned across the application. This is particularly important when components are supplied by different developers. In the preceding example, the developer who writes the code that refers to the role called `manager` has no way to know that there will be a role called `manager` in the deployment system. Only the deployer knows the name of the role, and might have made a decision to call this role `administrator`. At deployment time, the deployer will map the coded role `manager` onto the target role `administrator`.

EJB-Tier Security API

The API in the EJB tier is similar to the API in the web tier. However, the EJB components have *callers* instead of users, and the method names reflect this difference.

The security context of the current caller is made available to the EJB component by its container through the `EJBContext` objects. An EJB component typically saves its context object in an instance variable so it can be used later, as the snippet in Code 15-4 demonstrates.

Code 15-4 EJB Component Context Object Saved in an Instance Variable

```
package com.example;

import javax.annotation.Resource;
import javax.annotation.security.DeclareRoles;
import javax.annotation.security.RolesAllowed;
import javax.ejb.SessionContext;
import javax.ejb.Stateless;

@Stateless
@DeclareRoles({"admin", "user"})
public class SecurityBean {

    @Resource SessionContext ctx;

    @RolesAllowed(value = {"admin", "user"})
    public String getText() {
        String name = ctx.getCallerPrincipal().getName();
        return name + ", your admin status is: " +
          ctx.isCallerInRole("admin");
    }
}
```

There is also an `isCallerInRole` method, which corresponds to the web component's `isUserInRole` method, and is subject to the same deploy-time name resolution.

Note – Message-driven beans do not have callers in the same sense as session beans. You cannot determine the identity of the client that posted the message to the message-driven bean's queue. Therefore, the security API is of limited value in message-driven beans.

Configuring Authentication in the Web Tier

A detailed description of web-tier authentication is beyond the scope of this course. The following sections provide a basic outline of web-tier authentication.

Selecting the Authentication Type

The Java EE specification recognizes three types of authentication in the web tier. Each has its own configuration requirements.

- HTTP Basic – The only configuration necessary or possible is to select this authentication method by specifying it in the `web.xml` deployment descriptor. You can also use the IDE to perform this task.
- Client Certificate – The only configuration necessary or possible is to select this authentication method by specifying it in the `web.xml`. You also need to configure the web container's certificate database and allocate appropriate trust levels, but this process is specific to the particular application server.
- Form-based – The only configuration necessary or possible is to select this authentication method by specifying it in the `web.xml`. In addition, you must create a login page and an error page in the HTML or JSP software. These are the pages that are presented by the web container in response to a request that requires authentication. The pages can have any names. Use the `web.xml` to specify the names to the container.
- Programmatic – Role your own authentication. The Servlet 3.0 specification (Java EE 6) allows username and password collection using application specific methods.

Code 15-5 Selecting the authentication type in Java EE 6 with a `web.xml` deployment descriptor.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
    <login-config>
        <auth-method>BASIC</auth-method>
        <realm-name/>
    </login-config>
</web-app>
```

Note – The default authentication type is HTTP Basic. No `web.xml` file is needed to use authentication in Java EE 6.

Creating an HTML Login Page for Form-Based Authentication

The login page can have any appearance or layout, but the names of the form elements must comply with the specification. In practice, of course, you usually create login and error pages that match the look and feel of the rest of the application. The use of a JSP component or a servlet for the login page allows it to contain dynamic content, if necessary. The snippet in Code 15-6 shows a minimal section of HTML.

Code 15-6 Minimal Section of HTML

```
<form action="j_security_check" method="post">
<-- layout code not shown for clarity -->
Username:
<input name="j_username"/><br/>
Password:
<input name="j_password" type="password"/><br/>
<input type="submit" name="submit" value="Log in"/>
</form>
```

The HTML layout and presentation is unimportant to the authentication process. What matters is that the `action` attribute should be `j_security_check`, and the username and password fields should have the names `j_username` and `j_password`, respectively. It is these names that signal to the web container that authentication data is being supplied.

HTTP Security and JAX-WS Clients

Because a JAX-WS-based web service client uses HTTP for communication, it is affected by any authentication settings on the web server that provides the web service. You can provide a user name and password programmatically with JAX-WS.

Code 15-7 JAX-WS Supplied User Name and Password

```
SayHelloService service = new SayHelloService();
SayHello port = service.getSayHelloPort();

((BindingProvider)port).getRequestContext().put(BindingProvider.USERNAME_PROPERTY,
"username");
((BindingProvider)port).getRequestContext().put(BindingProvider.PASSWORD_PROPERTY,
"password");
```

The user name and password supplied to a JAX-WS client is sent to the web server when calling the methods of a web service. Before using the methods of a web service, you must have access to the service's WSDL file. If the WSDL file is password protected, you must configure a JAX-WS client with a local copy.

Summary

The security model of the Java EE platform centers on providing a declarative security model where the container does the work. The developer or application assembler specifies which methods of EJB components, or which URL patterns, are accessible to various user roles. Authentication, encryption, and certificate management are handled internally by the containers, which leaves you free to concentrate on the access policy.

Appendix A

EJB Technology in the Past

This appendix provides an explanation of the different terms for EJB technology that are used by older specifications. An understanding about how the terms have come to be used might make it easier to reconcile the logical ambiguities that you might have about EJB technology terms used today.

Historical Development of EJB Technology

Developers who are new to EJB technology are often confused by the conflicting and inconsistent nomenclature that they encounter for the various elements of an EJB component. You should understand that EJB technology developed iteratively from RMI technology, and new terms were added to the vocabulary without changing the earlier terms. This allowed developers to go on using the same terms for elements with which they were familiar, but did in the end led to logical inconsistencies.

In time, this problem will go away, as the nomenclature set out in the EJB 3.0 specification is entirely logical. However, not all writers use the new terminology and, because anything written before the EJB 3.0 specification will use the old terminology, it is important to be familiar with the meanings of the old terms as well.

In traditional Java technology-based RMI (Java RMI), a client's access to the *remote object* was controlled by a *remote interface*. Early versions of the EJB architecture extended Java RMI in two ways:

- An EJB was provided with a *home object* to act as a factory for its instances.
- The EJB implementation class itself was concealed from clients, by being hidden behind a proxy on the server side.

The client's remote interface now gave access to the proxy, not to the implementation class itself. Initially the proxy was known as the *remote object*, as in Java RMI, but this term was dropped in favor of *EJB object*. The reason for this was that it was the EJB implementation class itself that was felt to be analogous to the remote object in traditional RMI, not the proxy.

So, by the time that the EJB 1.1 specification was ratified, the names of the EJB elements had settled as follows. Each EJB component had:

- A *home interface* that provided access to a *home object*
- A *remote interface* that provided access to an *EJB object* (not a *remote object*)

At this time, the EJB architecture was exclusively a distributed architecture, and the problem of what to call the *remote object* when the client and the EJB component were not actually remote simply did not arise.

EJB 2.x Specification

The EJB 2.0 specification significantly changed the EJB architecture by adding support for local client views. This caused problems with nomenclature because the interface that controlled access to the EJB component's business methods was called the *remote interface*, which was inappropriate when the client was, in fact, not remote from the EJB component. So the EJB 2.0 specification added a new term, *local interface*.

The local interface was the equivalent of the remote interface, but was used by local clients. The home interface also gained a local version, the *local home interface*. Logically, if there was a *home interface* and a *local home interface*, then there should have been a *remote interface* and a *local remote interface*. Clearly, local remote interface is an oxymoron, which accounts for why the term *local interface* was chosen instead.

The problem was that this naming did not give a single term by which the local interface and remote interface could be referred to collectively. If writers wanted to discuss the home interfaces collectively, they could just say *home interface*, without specifying whether it was local or remote. However, to refer to the local interface and remote interface collectively, writers had to say local and remote interface every time. Worse, a convention arose of using the term *remote interface* to mean local and remote interface. Because remote interfaces are used more frequently than local interfaces, and because it is usually clear from the context of the sentence which usage of remote interface was meant, this did not cause too much trouble for experienced developers. However, this convention did cause trouble for novices and for authors of training materials.

Moreover, the local equivalents of the home and remote interfaces were associated with different versions of the server-side proxies. The home object retained the name *home object*, but qualified by the prefixes *local* and *remote* where necessary. However, the EJB object did not split into a local EJB object and a remote EJB object. Instead, the EJB object became either an EJB object or a *local object*.

In summary, in the EJB 2.0 specification, the term *home interface*, means either the remote home interface or the local home interface, and the term *remote interface* means one of two things:

- Remote interface could mean the true remote interface.
- Remote interface could also mean either the local interface or the remote interface.

The proxies on the server were the local and remote home object, the EJB object, and the local object.

This confusion of names was corrected in the EJB 2.1 specification. In this version of the EJB specification, the EJB component has two interface types:

- The home interface (as before)
- The (new) *component interface*

Each of these can be qualified by *local* or *remote*.

On the server side of the communication there are a home object, and a *component object*. Again, each can be qualified by local and remote, accordingly. This nomenclature is much more logical. A local home interface controls access to a local home object. A local component interface controls access to a local component object. A remote component interface controls access to a remote component object. And so on. If writers want to refer to the component interface without specifying whether it is the local or remote version, they can use the term *component interface* on its own.

There is one remaining complication. Although the server-side representation of the component interface is formally the component object, many writers preferred to use the term EJB object, because it is familiar and does not cause any confusion in practice.

Acronyms

A

API	application programming interface
ASP	Active Server Pages

B

B2B	business-to-business
B2C	business-to-consumer
BMP	bean-managed persistence
BMT	bean-managed transaction

C

CGI	Common Gateway Interface
CMP	container-managed persistence
CMR	container-managed relationships
CMT	container-managed transaction
CORBA	Common Object Request Broker Architecture
CSIv2	CORBA secure interoperability, version 2

D

DAO	data access object
DTD	document type definition

DAO	data access object
DBMS	database management system
DTO	data transfer object

E

EAR	enterprise archive
ebXML	Electronic Business Extensible Markup Language
EIS	enterprise information system
EJB™	Enterprise JavaBeans™
ERP	enterprise resource planning

G

Gof	Gang of four
GUI	graphical user interface

H

HTML	Hyper-Text Markup Language
HTTP	Hypertext Transport Protocol

I

IDE	integrated development environment
***the* IDE**	Netbeans™ IDE
IDL	interface definition language
IIOP	Internet Inter-ORB Protocol
IP	Internet Protocol

J

Java EE™	Java™ Platform, Enterprise Edition
Java SE™	Java Platform, Standard Edition
JAAS	Java Authentication and Authorization Service
JAF	JavaBeans™ Activation Framework
JAR	Java Archive
JAXM	Java API for XML Messaging
JAXP	Java API for XML Processing
JAX-RPC	Java API for XML-based remote procedure call
JAX-RS	Java API for RESTful web services
JAX-WS	Java API for XML-based web services
JCP^{SM}	Java Community Process^{SM}
JDBC™	Java DataBase Connectivity™
JDO	Java Data Objects
JMS	Java Message Service
JMX™	Java Management extensions
JNDI	Java Naming and Directory Interface™
JRMP	Java Remote Method Protocol
JSF™	JavaServer Faces™
JSP™	JavaServer Pages™
JSTL	JavaServer Pages Standard Tag Library
JTA	Java Transaction API
JVM™	Java Virtual Machine

L

LDAP	Lightweight Directory Access Protocol
LRU	least recently used

M

MDB	message-driven bean
MVC pattern	Model-View-Controller pattern

N

NIS+	Network Information Service Plus

O

OMG	Object Management Group
ORB	Object Request Broker
OTS	Object Transaction Service

P

PK	primary key
POJO	Plain Old Java Object

Q

QL	query language

R

RAR	resource (adapter) archive
RMI	Remote Method Invocation
RPC	remote procedure call

S

SOAP	Simple Object Access Protocol
SQL	structured query language
SSL	secure sockets layer

T

TCP	Transmission Control Protocol
TLD	tag library descriptor
TLS	transport layer security

U

UML	Unified Modeling Language
URI	Uniform Resource Identifier
URL	Universal Resource Locator

W

WAF	Web Application Framework
WAR	web archive
WSDL	web services description language
WWW	World Wide Web

X

XML	Extensible Markup Language
XSLT	XML Stylesheet Language for Transformations