



Integrated Cloud Applications & Platform Services



Developing Applications for the Java EE 7 Platform

Student Guide

D98815GC10

Edition 1.0 | February 2018 | D100413

Learn more from Oracle University at education.oracle.com

ORACLE®

Author

Vasily Strelnikov

**Technical Contributors
and Reviewers**

Phil Franklin

Girija C

Daniel Milne

Dorin Paraschiv

Jacobo Marcos

Nikolai Elistratov

Editors

Raj Kumar

Vijayalakshmi Narasimhan

Graphic Designer

Kavya Bellur

Publishers

Syed Ali

Pavithran Adka

Raghunath M

Asief Baig

Srividya Rameshkumar

Sumesh Koshy

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Disclaimer

This document contains proprietary information and is protected by copyright and other intellectual property laws. You may copy and print this document solely for your own use in an Oracle training course. The document may not be modified or altered in any way. Except where your use constitutes "fair use" under copyright law, you may not use, share, download, upload, copy, print, display, perform, reproduce, publish, license, post, transmit, or distribute this document in whole or in part without the express authorization of Oracle.

The information contained in this document is subject to change without notice. If you find any problems in the document, please report them in writing to: Oracle University, 500 Oracle Parkway, Redwood Shores, California 94065 USA. This document is not warranted to be error-free.

Restricted Rights Notice

If this documentation is delivered to the United States Government or anyone using the documentation on behalf of the United States Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS

The U.S. Government's rights to use, modify, reproduce, release, perform, display, or disclose these training materials are restricted by the terms of the applicable Oracle license agreement and/or the applicable U.S. Government contract.

Trademark Notice

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Contents

1 Course Introduction

- Course Objectives 1-2
- Audience 1-3
- Class Introductions 1-4
- Course Environment 1-5
- Course Structure 1-6
- Course Practices 1-7
- Course Appendices 1-8
- Course Schedule 1-9
- Summary 1-10

2 Introduction to Java EE

- Objectives 2-2
- Requirements of Enterprise Applications 2-3
- Separation of Business Logic from Platform Services 2-4
- Structure and Purpose of Java EE 7 Server, Containers, and APIs 2-5
- EJB Lite and EJB Full Containers 2-7
- Evolution of Web Design 2-8
- MVC (Model View Controller) 2-9
- Java EE Web Container Components: Servlets 2-10
- Java EE Web Container Components: JSPs 2-11
- Java EE Web Container Components: JSFs 2-12
- Java EE Web Container Components: REST Services 2-13
- Java EE Web Container Components: Web Sockets 2-14
- Java EE 7 Web Services 2-15
- Java EE 7 Business Logic Handling Technologies 2-16
- Maintaining Application State 2-17
- Session EJB Types 2-18
- Message-Driven EJB 2-20
- Assembling Application Components with CDIs 2-21
- JSF Managed Beans, CDI Beans, EJBs 2-22
- Request Scope 2-23
- Session Scope 2-24
- Application Scope 2-25
- View Scope 2-26

| | |
|---|------|
| Conversation Scope | 2-27 |
| Dependent Scope | 2-28 |
| Injecting Beans | 2-29 |
| Java EE Packaging and Deployment | 2-30 |
| Annotations or Deployment Descriptors | 2-31 |
| Annotations with Deployment Descriptors | 2-33 |
| Java Naming Directory Interface Objects | 2-34 |
| Container-Managed Injections | 2-36 |
| JDNI Lookups | 2-37 |
| Summary | 2-38 |
| Practices | 2-39 |

3 Managing Persistence by Using JPA Entities

| | |
|---|------|
| Objectives | 3-2 |
| Java Persistence API | 3-3 |
| JPA Entities: Basics | 3-5 |
| Persistent Field Versus Persistent Property | 3-8 |
| Using Access Annotation | 3-9 |
| Converters | 3-10 |
| Generated Keys | 3-11 |
| JPA Lifecycle Callback Methods | 3-13 |
| Validating Entities | 3-14 |
| Using Bean Validation Constraints | 3-16 |
| Container Managed Persistence | 3-18 |
| Locally Managed Persistence | 3-21 |
| Entity Manager Operations | 3-23 |
| Locking and @Version | 3-25 |
| Changing Locking Mode | 3-27 |
| Java Persistence Query Language (JPQL) | 3-28 |
| Using JPQL with non-Entity classes | 3-30 |
| Executing JPQL Statements | 3-31 |
| Summary | 3-33 |
| Practice Overview | 3-34 |

4 Implementing Business Logic by Using EJBs

| | |
|-----------------------------------|-----|
| Objectives | 4-2 |
| EJBs and EJB Container | 4-3 |
| Enterprise JavaBean Types | 4-4 |
| Session EJBs | 4-5 |
| Accessing Session Beans | 4-7 |
| Stateless Session Bean Life Cycle | 4-9 |

Stateful Session Bean Life Cycle 4-10
Singleton Session Bean Life Cycle 4-11
Asynchronous EJB operations 4-13
Java Transaction API 4-14
Programmatic Transactions (BMT) 4-15
Declarative Transactions (CMT) 4-16
Demarcate Transactional Attributes 4-17
Transaction Scoped Beans 4-18
Timers 4-19
Calendar-Based Timer Expressions 4-21
Define Programmatic Timers 4-23
Define Automatic Timers 4-25
Manage Timers 4-26
Define Interceptors 4-27
Types of Interceptors 4-29
Apply Interceptors 4-30
Summary 4-31
Practice Overview 4-32

5 Using Java Message Service API

Objectives 5-2
Java Message Service (JMS) API 5-3
JMS Destination Types 5-5
JMS 2.0 API 5-7
JMS Context 5-9
Java SE Message Producer 5-11
Java SE Message Consumer 5-13
Java SE Asynchronous Producers and Consumers 5-15
JMS Session Modes and Message Acknowledgments 5-16
Handle JMS Messages 5-18
Java EE Message Producer 5-19
Java EE Message Consumer 5-20
Topics Shared/Unshared Subscriptions 5-21
Queue Message Brower 5-22
Message-Driven Bean (MDB) 5-23
MDB Life Cycle 5-25
JMS and Transactions 5-26
Handle Errors with Transactions 5-27
Summary 5-28
Practice Overview 5-29

6 Implementing SOAP Services by Using JAX-WS

- Objectives 6-2
- WebServices and SOAP 6-3
- Web Service Interaction Patterns 6-4
- Web Service Interface 6-5
- XML Schema Definition 6-6
- WSDL Schemas and Namespaces 6-8
- WSDL Messages, PortTypes, and Operations 6-9
- WSDL Bindings and Services 6-10
- Top-down versus Bottom-up approach 6-12
- Map Java Interface to WSDL 6-13
- JAX-WS Implementation 6-15
- Create JAVA JAX-WS Client 6-16
- Invoke SOAP Service from Java SE Client 6-19
- Invoke SOAP Service from Java EE Component 6-20
- Summary 6-21
- Practice Overview 6-22

7 Creating Java Web Applications by Using Servlets

- Objectives 7-2
- HTTP Protocol Basics: Sending Requests 7-3
- HTTP Protocol Basics: Getting Responses 7-4
- Create Servlet 7-5
- Override Servlet Request Handling Operations 7-7
- Provide Request Handling Logic 7-8
- Retrieve Request Headers 7-9
- Retrieve Parameters 7-10
- Use Cookies 7-11
- Produce Different Content Types 7-12
- Manage Servlet Life Cycle with Container Callbacks 7-13
- CDI Beans 7-14
- HTTP Session Tracking 7-15
- Web Container Life Cycle Events 7-16
- Request Dispatcher 7-19
- Servlet Filters 7-21
- Asynchronous Servlets 7-23
- Nonblocking I/O 7-25
- Handle Errors 7-28
- Summary 7-29
- Practice Overview 7-30

8 Creating Java Web Applications by Using JSPs

- Objectives 8-2
- Create Java Server Page 8-3
- Java Server Page Syntax 8-4
- Java Server Page XML Syntax 8-6
- Expression Language 8-7
- Expression Language Operators 8-8
- JSP Scopes and Implicit Objects 8-9
- Use CDI Beans in JSPs 8-10
- Standard Tag Library (JSTL) 8-11
- Create JSP Error Handlers 8-13
- Summary 8-14
- Practice Overview 8-15

9 Implementing REST Services using JAX-RS API

- Objectives 9-2
- REST Service Conventions and Resources 9-3
- REST Communication Model 9-4
- Implementing REST Services using JAX-RS API 9-5
- Mapping Resources to URI Paths 9-7
- Mapping REST Resource Operations 9-8
- Handling Different Media Types 9-9
- Passing Parameters 9-11
- Validating Values 9-12
- Handling Web Service Errors 9-13
- Asynchronous REST Services 9-15
- Asynchronous EJB and REST Services 9-17
- Invoking REST Service from JavaScript Client 9-18
- Invoking REST Service from Java Client 9-19
- Invoking REST Service from Asynchronous Java Client 9-20
- Summary 9-21
- Practice 9-22

10 Creating Java Applications with WebSockets

- Objectives 10-2
- WebSockets Network Protocol 10-3
- WebSocket Life Cycle 10-4
- Defining WebSocket Endpoints 10-5
- Using PathParam Annotation 10-7
- Using WebSocket Session 10-8

| | |
|--|-------|
| Using RemoteEndpoint Objects | 10-10 |
| Encode and Decode Messages | 10-11 |
| Handle WebSocket Messages | 10-12 |
| Handle WebSocket Errors | 10-14 |
| Encoding and Decoding WebSocket Messages | 10-15 |
| Implementing WebSocket Message Encoder | 10-16 |
| Implementing WebSocket Message Decoder | 10-17 |
| Creating JSON Messages | 10-18 |
| Parsing JSON Messages | 10-19 |
| Invoking WebSocket Server from a JavaScript Client | 10-20 |
| Invoking WebSocket Server from a Java Client | 10-21 |
| Summary | 10-22 |
| Practice | 10-23 |

11 Developing Web Applications Using JavaServer Faces

| | |
|---|-------|
| Objectives | 11-2 |
| JavaServer Faces Concepts | 11-3 |
| Faces Servlet Registration | 11-4 |
| JSF Configuration | 11-5 |
| JSF Facelet Structure | 11-6 |
| JSF Request-Response Lifecycle | 11-7 |
| JSF Libraries | 11-9 |
| JSF HTML Library UIComponents | 11-10 |
| JSF HTML Passthrough | 11-11 |
| Using Validators and Converters | 11-12 |
| JSF Templates | 11-13 |
| Describe JSF Navigation | 11-14 |
| Configuring Navigation Rules | 11-15 |
| Using Faces Flows | 11-16 |
| Action and ActionListener Attributes | 11-17 |
| Value and Binding Attributes | 11-18 |
| Using immediate attribute | 11-19 |
| Using FacesContext Object | 11-20 |
| JSF Localization | 11-21 |
| Displaying Messages | 11-22 |
| Producing Messages From CDI Beans | 11-23 |
| Using Managed Properties | 11-25 |
| Adding AJAX code to Facelets | 11-26 |
| Extended JSF Frameworks and Component Libraries | 11-27 |
| Summary | 11-28 |
| Practice | 11-29 |

12 Securing Java EE Applications

- Objectives 12-2
- JAAS Security Concepts 12-3
- JAAS Configuration 12-5
- Request Authentication and Authorization 12-7
- Login Module Configuration 12-8
- Programmatic Authentication 12-10
- Declare Application Roles 12-11
- Define Security Constraints 12-12
- Java EE Programmatic Security 12-14
- Web Service Security 12-15
- WS-Security 12-16
- Summary 12-17
- Practice 12-18

A Java Logging

- Objectives A-2
- Java Logging Frameworks A-3
- Using Java Logging API A-4
- Logging Method Categories A-5
- Guarded Logging A-7
- Log Writing Handling A-8
- Logging Configuration A-10
- Application Server Logging Configuration A-11
- Configuring the WebLogic Logging Service A-12
- Viewing WebLogic Server Logs A-13

B CDI Beans

- Objectives B-2
- Using Named Qualifiers B-3
- Using Custom Qualifiers B-4
- Using Alternative Qualifiers B-5
- Producer and Disposer Methods B-6
- Interceptors B-7
- CDI Events B-8
- Stereotypes B-9

C BeanValidation and JPA API

- Objectives C-2
- Custom BeanValidation Constraints C-3

Entity Relationship Types C-4
Mapping Entity Relationships C-6
Entity Relationship Mapping Properties C-7
Mapping Embeddable Classes C-8
Mapping an Entity to Multiple Tables C-9
Composite Primary Key C-11
Using Inheritance with Entities C-13
Using Unmapped Superclass C-14
Using Mapped Superclass C-15
Entity Inheritance Mapping Strategies C-16

D Batch and Concurrency APIs

Objectives D-2
Concurrency D-3
Executor Service D-5
Managed Task Listener D-6
Batch Processing: Overview D-7
Job Specification Language (JSL) XML Structure D-8
Batch API Structure D-12
Describe Job Using JSL XML Document D-14
Run Batch Job D-15

E JAXB API

Objectives E-2
JAXB API E-3
Bind Java Classes to XML Schema E-4
Read and Write XML with JAXB E-5
JAXB Annotations Part I E-6
JAXB Annotations Part II E-7

F "Pre-CDI" Servlet Examples

Objectives F-2
Using Request Application Attributes Without CDI F-3
Using HttpSession Attributes Without CDI F-4

Course Introduction



ORACLE®



1

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Jairo Jose Silvera Sarmiento (jairo.silvera@gmail.com) has a
non-transferable license to use this Student Guide.

Course Objectives

In this course, you will gain experience with a broad array of Java EE technologies.

The enterprise technologies that are presented in this course include:

- Handling business logic using POJOs, EJBs, SOAP WebServices, and JMS
- Managing persistency using JPA entities
- Creating Java web applications using servlets, JSPs, JSFs, REST Services, and WebSockets
- Securing Java EE applications
- Deploying Java EE applications



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Audience

The target audience includes those who have:

- Completed the Java SE 8 Programming course
- Obtained the Java SE 8 Oracle Certified Professional certification
- Experience with the Java language
- Experience with XML
- Experience with basic database concepts and a basic knowledge of SQL

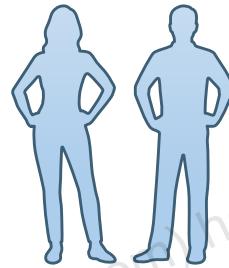


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Class Introductions

Briefly introduce yourself:

- Name
- Title or position
- Company
- Experience with Java programming
- Reasons for attending



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Course Environment

JDK 8

NetBeans 8.1

WebLogic Server 12c (12.2.1.2)

Derby Java Database



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Course Structure

- Understand the Java EE Platform
 - Explain Purpose of Java EE Containers, APIs and Services
 - Deploy Java EE Applications
 - Interconnect Application Components with CDI Annotations and JNDI
- Develop Java EE Application Back-Ends:
 - Manage Persistence with JPA API
 - Manage Business Logic with EJB
 - Produce and consume messages with JMS API
 - Expose SOAP WebServices with JAX-WS API
- Develop Java EE Web Application User Interfaces by using:
 - Servlet
 - JSP
 - JSF
 - REST Services using JAX-RS API
 - WebSockets
- Secure Java EE Applications



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Course Practices

During this course practice sessions you develop product management application.

This application is going to start as a simple client-server application, but will evolve into a Java Enterprise Application having following components:

- Java Persistence API components to handle product database objects
- Enterprise Java Beans components to handle product management application business logic
- Java Message Service API components to produce and consume messages
- SOAP WebService to produce product quotes
- Web user interface to search, display and update products designed with
 - Servlets
 - Java Server Pages
 - Java Server Faces
- REST Service to check product discount
- WebSockets application to implement chat between users
- You will also secure this application using both programmatic and declarative approaches

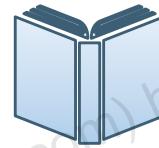


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Course Appendices

- Appendix A: Java Logging API
- Appendix B: Advanced CDI Beans
- Appendix C: Bean Validation and JPA API
- Appendix D: Batch and Concurrency APIs
- Appendix E: Using JAXB API
- Appendix F: "Pre-CDI" Servlet Examples



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Course Schedule

Lesson 1: Course Introduction
Lesson 2: Introduction to Java EE
Lesson 3: Managing Persistence by Using JPA Entities

Day One

Lesson 4: Implementing Business Logic by Using EJBs
Lesson 5: Using Java Message Service API
Lesson 6: Implementing SOAP Services by Using JAX-WS

Day Two

Lesson 7: Creating Java Web Applications by Using Servlets
Lesson 8: Creating Java Web Applications by Using JSPs

Day Three

Lesson 9: Implementing REST Services using JAX-RS API
Lesson 10: Creating Java Applications with WebSockets

Day Four

Lesson 11: Developing Web Applications Using JavaServer Faces
Lesson 12: Securing Java EE Applications

Day Five



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

This schedule is for guidance purposes only. Depending on the pace of practices exact timings may vary.

Summary

In this lesson, you have looked at the overall course structure and objectives.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Introduction to Java EE



ORACLE®



2

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Jairo Jose Silvera Sarmiento (jairo.silvera@gmail.com) has a
non-transferable license to use this Student Guide.

Objectives

This lesson gives an overview of Java EE 7 Architecture and its components. After completing this lesson, you should be able to describe:

- Standards, containers, APIs, and services
- Application component functionalities mapped to tiers and containers
 - Web container technologies
 - Business logic implementation technologies
 - Web service technologies
- Packaging and deployment
- Enterprise JavaBeans, managed beans, and CDI beans
 - Understanding lifecycle and memory scopes
- Linking components together with annotations, injections, and JNDI



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Requirements of Enterprise Applications

The Java EE platform:

- Is an architecture for implementing enterprise-class applications
- Uses Java and Internet technology
- Has a primary goal of simplifying development of enterprise-class applications through an application model that is:
 - Vendor-neutral
 - Component-based



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Java EE is a collection of specifications that require an implementation. For a full list of application servers that are certified as Java EE compatible, see the website at:

<http://www.oracle.com/technetwork/java/javaee/overview/compatibility-jsp-136984.html>.

Separation of Business Logic from Platform Services



Developer's Checklist

- Business services
- Persistence and Transaction management
- Multithreading
- Security management
- Networking

Developer's Checklist

- Business services

Services Provided by Server

- Persistence and Transaction management
- Multithreading
- Security management
- Networking



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

A key feature of the Java EE Platform is the strict separation of application components from the general services and infrastructure. One of the main benefits of the Java EE Platform for application developers is that it makes it possible 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 if there is a failure. The application server infrastructure and services are responsible for these functions.

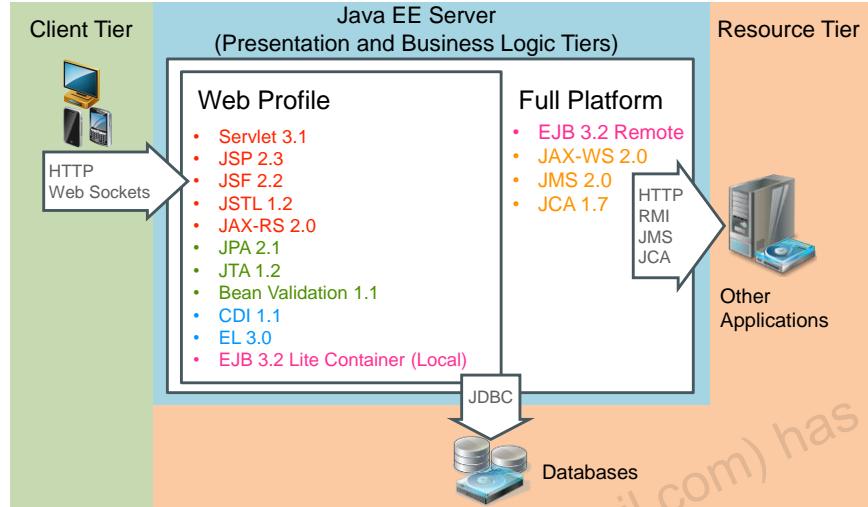
The Java EE server provides infrastructure services to help developers to create multiuser enterprise applications. These include:

- User authentication and authorization services to implement security requirements
- Persistence and transaction management
- Concurrency and multithreading
- Networking, connectivity, and distributed object management
- Messaging services
- Resource management, object life cycle, and naming services

Structure and Purpose of Java EE 7 Server, Containers, and APIs

The Java EE platform describes Web and EJB containers and various APIs:

- Web Container Technologies
- Java SE Technologies
- Technologies in all containers
- EJB Container Technologies
- Technologies supported with Full Platform server implementation



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

A Java EE server is used to implement the presentation and business logic application tiers.

Java EE 7 includes profiles for application developers that do not need the full Java EE Platform but require application portability. Developers can choose between:

- The Java EE Web Profile: Focused on web applications and supporting technologies
- The Java EE Full Platform: Complete Java EE application servers

Note: Technically “Full Platform” is not a profile. The only profile included in the Java EE specification is the web profile. Additional profiles may be added in future releases.

The complete list of technologies relevant to the Java EE 7 specifications can be found here:

<http://www.oracle.com/technetwork/java/javaseee/tech/index.html>

You can download (free of charge) the specifications that define the Java EE Platform from the Java Community Process (JCP) website (<http://jcp.org>). JCP enables individuals and organizations to become involved in the future development of Java. Involvement can range from reviewing early draft specifications to joining expert groups that write new Java specifications. For more on becoming involved in the JCP, see the website at <http://jcp.org/en/participation/overview>.

The specifications outline the rules that 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.

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. The Java EE server vendors use these specifications when they develop server elements.

<https://www.jcp.org/en/jsr/detail?id=342>

Some JSRs define only APIs, whereas others also define the services that must be supplied by compatible implementations.

APIs: These include a collection of classes, annotations, and interfaces that developers can use. An API is packaged into a library, which can be added to a Java application.

JavaMail is an example of a downloadable library, which can be added to any Java application.

Services: JSRs may outline a required environment in which the developer-supplied components will execute. A container is a part of an application server, which supplies the required runtime environment.

Servlets are an example of components that require a (web) container to execute.

EJB Lite and EJB Full Containers

EJB Lite features:

- Required by the Web Profile
- Session beans components:
 - Stateless
 - Stateful
 - Singleton
- Support local clients
- Method invocations:
 - Synchronous
 - Asynchronous
- Transaction modes:
 - Container-managed
 - Bean-managed
- Declarative and programmatic security
- Automatically created EJB timers

EJB Full = EJB Lite + additional features:

- Required by Full Platform
- Message-driven beans
- Remote and local clients
- JAX-WS web service endpoints
- Persistent EJB timer service
- Support legacy services and EJB APIs



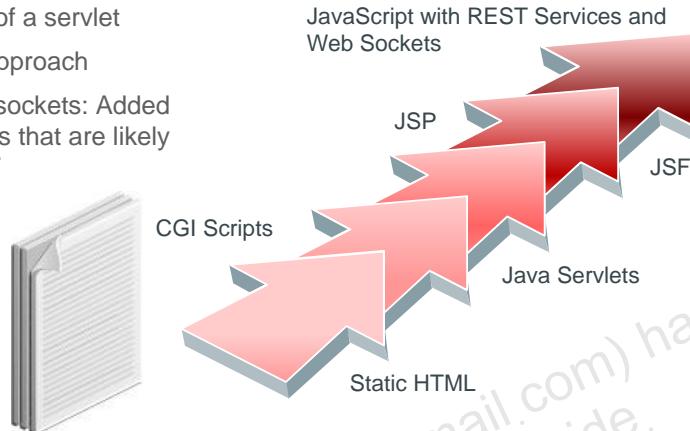
Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Components

| | EJB Lite | EJB Full |
|--|-----------------|-----------------|
| Session Beans (stateful, stateless, singleton) | Yes | Yes |
| Message-Driven Beans | No | Yes |
| 2.x/1.1 CMP/BMP Entity Beans | No | Yes |
| JPA 2.1 | Yes | Yes |
| Session Bean Client Views | | |
| Local/No Interface | Yes | Yes |
| 3.x Remote | No | Yes |
| 2.x Remote Home/Component | No | Yes |
| JAX-WS Web Services Endpoint | No | Yes |
| JAX-RPC Web Services Endpoint | No | Yes |
| Services | | |
| EJB Timer Service (non-persistent) | Yes | Yes |
| Asynchronous Session Bean Invocations (local) | Yes | Yes |
| Interceptors | Yes | Yes |
| RMI-IIOP Interoperability | No | Yes |
| Container-managed/Bean-managed Transactions | Yes | Yes |
| Declarative and Programmatic Security | Yes | Yes |
| Miscellaneous | | |
| Embeddable API | Yes | Yes |

Evolution of Web Design

- Web: Started as static HTML documents
- CGI scripts: Introduced dynamically generated content
- Java servlets: Multithreaded and scalable solution
- Java Server Pages: Improved UI design of a servlet
- Java Server Faces: Implemented MVC approach
- JavaScript with REST services and web sockets: Added client-side UI and event handling to pages that are likely to be produced by using Servlet/JSP/JSF

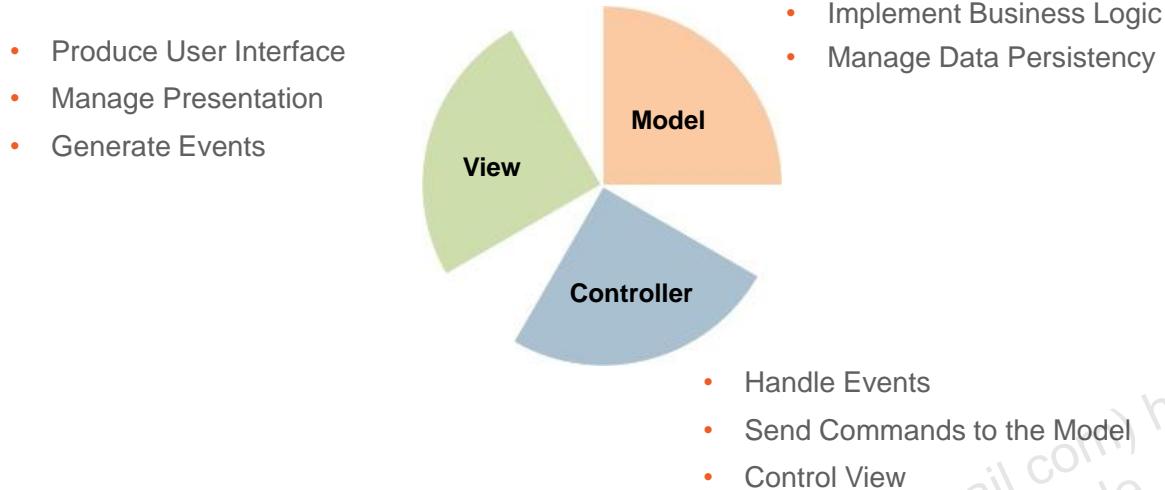


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Greater levels of dynamic processing and scalability were achieved as web design technology evolved. Simple static pages technologies evolved to first allow dynamic view generation, and then later improved the code structure by introducing separation of concerns by using patterns such as Model View Controller.

The latest stages of the Web Design evolution introduced dynamic processing on the client tier via JavaScript, which can access server-side functionalities via REST services and web sockets. However, your JavaScript code still has to be delivered to the client via more traditional means, such as a server-generated UI produced by JSPs, JSFs, or servlets, or even via static HTML document download.

MVC (Model View Controller)



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

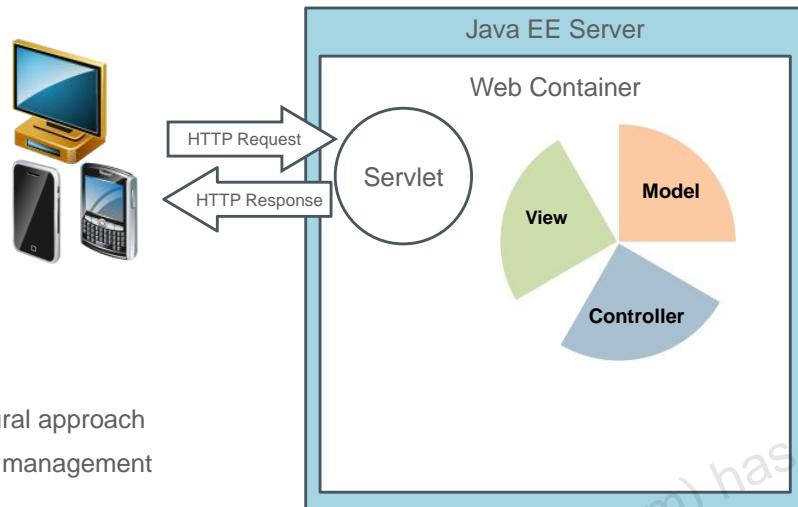
The Model View Controller design pattern addresses a separation of concerns design principle. It suggests that application code can be structured so each component in the application would be responsible for only one of the following aspects:

- Producing user interface presentation
- Handling user interface events
- Invoking application business logic

The benefits of this approach are as follows:

- The application code is more flexible. The same model (Business Logic) may be reused by different views (Presentations)—for example, web based or mobile app clients.
- Developers can specialize in the front-end or back-end logic of the system. For example, UI design or database persistence logic development requires very different skill sets.

Java EE Web Container Components: Servlets



Servlets:

- Are Java classes mapped to URLs
- Are typically invoked via HTTP
- Utilize request-response architectural approach
- Can mix business logic and layout management



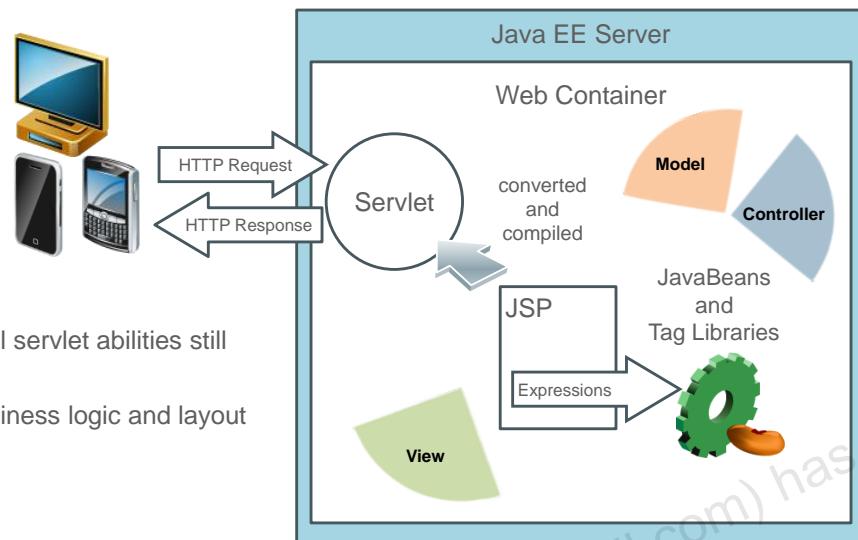
Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Historically Servlets were the first type of dynamic web user interface producing components in Java. Servlets do not require developers to adhere to the MVC pattern, allowing them to mix presentation and logic in the same class.

Java EE Web Container Components: JSPs

JSPs (Java Server Pages):

- Are turned into servlets (so all servlet abilities still apply)
- Offer better separation of business logic and layout management with:
 - JavaBeans
 - Tag libraries
 - Expression Language



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

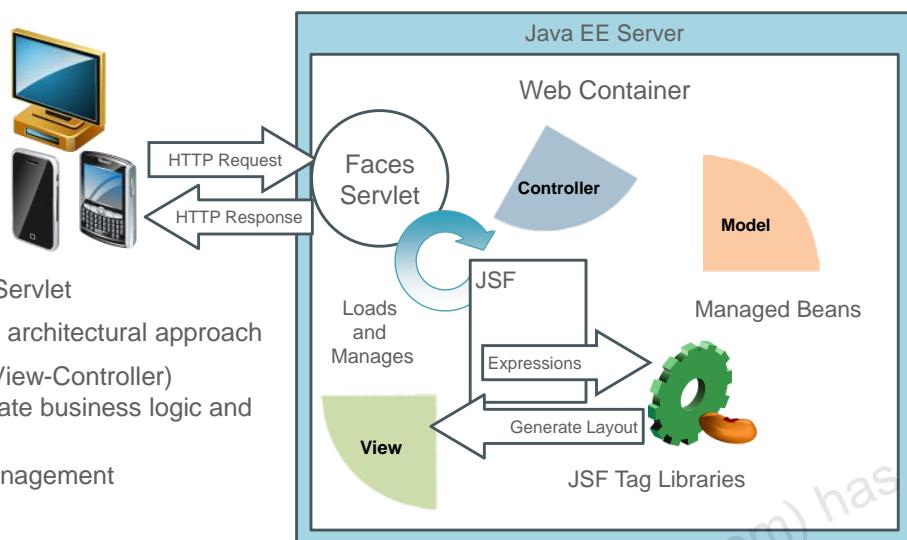
Java Server Pages give developers a mechanism to create Servlets in a way that separates user interface generation from the handling of events and logic.

The way developers work with Java Server Pages feels quite similar to the way HTML pages are designed. In fact, JSP pages can contain HTML markups, but they can also contain bits of Java code and references to Java objects. When Java Server Pages are deployed, they are turned into Servlets, so their execution and runtime capabilities are in no way different from a Servlet.

Java EE Web Container Components: JSFs

JSFs (Java Server Faces):

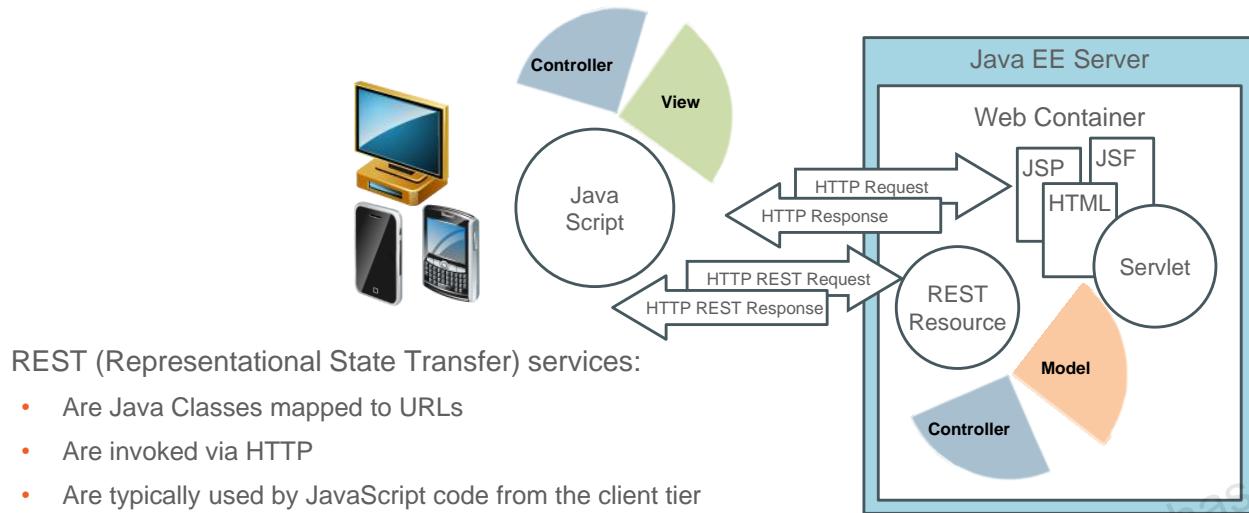
- Are interpreted by FacesServlet
- Utilize a component-base architectural approach
- Implement MVC (Model-View-Controller) architecture to fully separate business logic and layout management with:
 - JSF lifecycle management
 - Managed beans
 - Expression Language



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The Java Server Faces technology provides an API for representing components and managing their state. It defines an event handling model that allows server-side validations and data conversions. The JSF runtime controls page navigation. The JSF tag libraries implement components that developers can add to web pages to produce a layout.

Java EE Web Container Components: REST Services



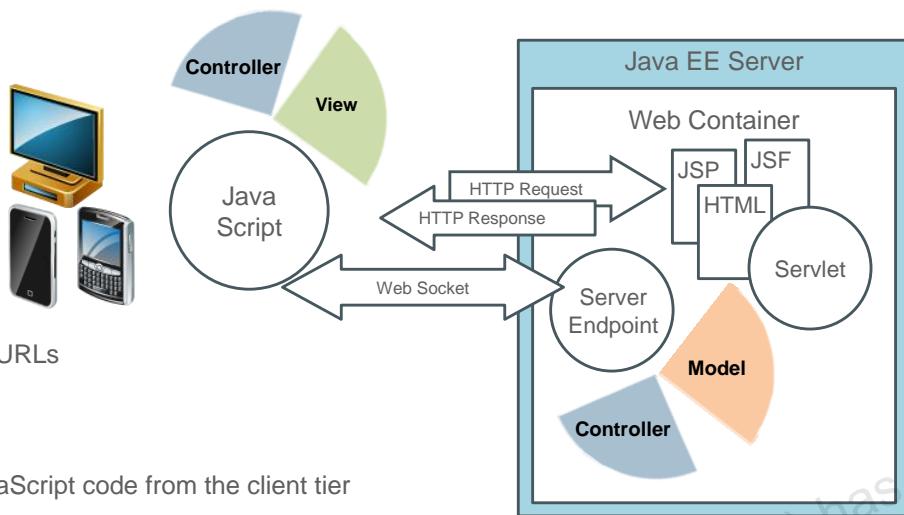
Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Representational State Transfer (REST) is an architectural style that specifies constraints such as the uniform interface, which, if applied to a web service, induce desirable properties, such as performance, scalability, and modifiability, which enable services to work best on the web. In the REST architectural style, data and functionality are considered resources and are accessed by using Uniform Resource Identifiers (URIs), typically links on the web. The resources are acted upon by using a set of simple, well-defined operations. The REST architectural style constrains an architecture to a client/server architecture and is designed to use a stateless communication protocol, typically HTTP. In the REST architecture style, clients and servers exchange representations of resources by using a standardized interface and protocol.

Java EE Web Container Components: Web Sockets

Web sockets:

- Java classes mapped to URLs
- Full duplex connection
- Can use server push
- Are typically used by JavaScript code from the client tier
- Present business-logic functionalities to the client
- Utilize component-base models



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Java EE 7 Web Services

All Web Services provide business functionalities in a way that disguises their implementation.



JAX-RS (REST services):

- Utilize HTTP protocol methods, such as:
 - GET
 - PUT
 - POST
 - DELETE
- Can transport any data, for example:
 - XML
 - JSON
- Typically used by browser and mobile UI



JAX-WS (SOAP services):

- Are transport protocol-independent
- Use standard WSDL descriptors
- Can transport XML described via XSD
- Provide a range of WS-* standard policies, such as:
 - WS-Security
 - WS-Reliability
 - WS-Addressing
 - WS-Transactions
- Typically used for system integration purposes and in SOA architecture



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Java EE 7 Business Logic Handling Technologies

CDI managed bean:

- Life cycle determined by memory scope context:
Request, View, Session, Application, Dependent, Conversation, and so on
- Can be invoked only locally

EJB (Enterprise JavaBean):

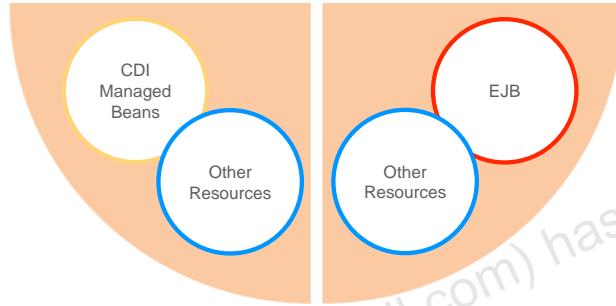
- Life cycle determined by the type of bean:
 - Session Beans: Stateless, Stateful, Singleton
 - Message Beans: Message-Driven
- Can be invoked locally or remotely

Other resources:

- EntityManager, JMS Queue or Topic, DataSource, EJBContext and so on represent container-managed resources.

Model implementation:

- Contains your business logic
- Contained within CDI managed beans
- or Enterprise JavaBeans
- or both



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



A CDI managed bean is implemented with a JavaBean class that is a simple POJO and may be used in a variety of contexts.

CDI beans are about loose coupling. They provide the ability to bind stateful components to well-defined but extensible lifecycle contexts and the ability to inject components into an application in a typesafe way, including being able to choose which implementation of an interface to inject at deployment time.

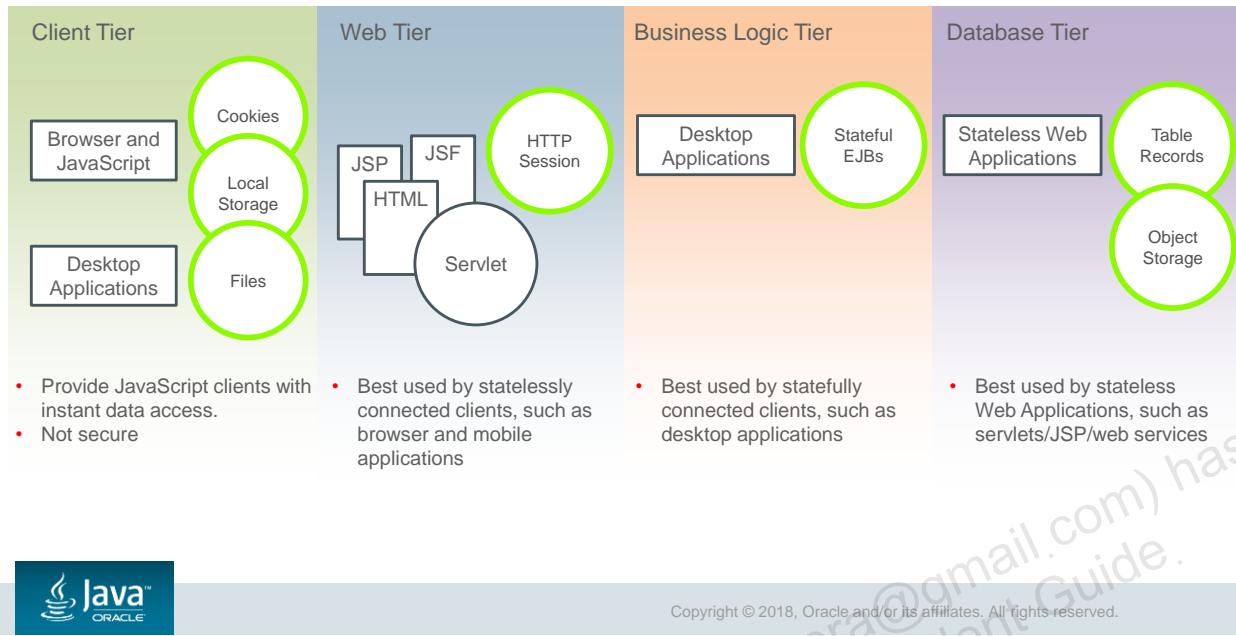
An enterprise bean is a server-side component that encapsulates the business logic of an application.

An EJB container provides system-level services to enterprise beans, the bean developer can concentrate on solving business problems. The EJB container, rather than the bean developer, is responsible for system-level services, such as life cycle, connectivity, transaction management, and security authorization.

Session EJB: Performs a task for a client; optionally, may implement a web service

Message EJB: Acts as a listener for a particular messaging type, such as the Java Message Service API

Maintaining Application State



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Applications may choose to maintain their state within different tiers. JavaScript and web browser type of applications may use cookies or local storage to record their state. Java Desktop clients may use files in a clients file system. This approach gives clients quick access to the required information, sometimes even without connectivity to other tiers. However, this approach is inherently not secure.

Web applications may use HTTP sessions and session-scoped beans to store the required state information, such as the context of a previous call made by a particular client. A good example of such approach would be storing Shopping Cart information in memory, on behalf of individual clients. This approach is much more secure compared to the client tier application state storage, but may require synchronizing session state across nodes in the cluster. This would imply an additional overhead when scaling the application.

Application code residing in the Business Logic tier can also be used to manage application state with stateful Enterprise JavaBeans. This approach requires the server to dedicate EJB instances to each client. This approach would create an unnecessary overhead if the client is not connected to the server permanently via protocols such as RMI. Because web browser and mobile clients usually connect to servers by using transient protocols such as HTTP, it would be more appropriate to use an HTTP session state management approach for such clients and utilize stateless session EJBs in the Business Logic tier.

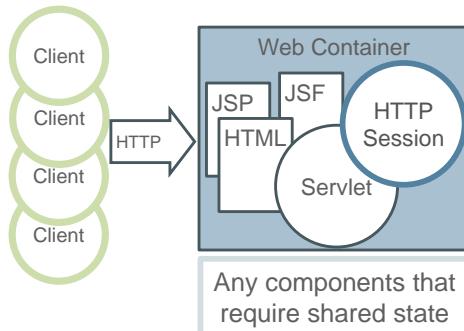
An application state may also be stored in the Database tier as table records, or big data object storage. This approach is typically used by web tier applications instead of the HTTP session approach. Recording state into the database may present an additional overhead compared to in-memory sessions. However, with the growth of application scale and the introduction of cluster deployments, HTTP session overhead is not a problem.

Session EJB Types

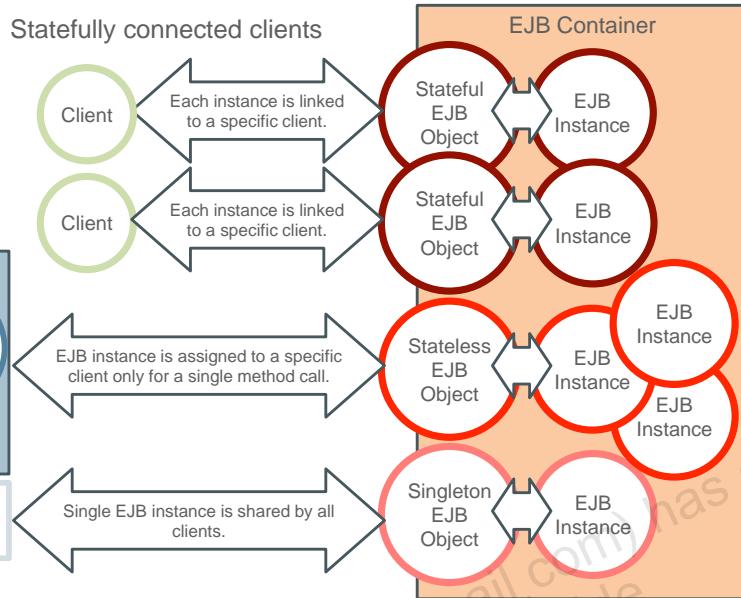
Clients that call EJBs may be:

- Desktop Java clients
- Browsers or mobiles via a web container
- Other EJBs

Statelessly connected clients



Statefully connected clients



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

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 the graphic in the slide.

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.

Stateless Session Beans

- The bean does not retain client-specific information.
- A client might not use the same session bean instance during subsequent method calls.
- A large number of clients can be handled at the same time.

Stateful Session Beans

- Beans belong to a particular client for an entire conversation.
- The client connection exists until the client removes the bean or the session times out.
- The container maintains a separate EJB object and EJB instance for each client.

There is always a cost to maintain client state; 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, stateful session beans are an effective way to store the client state.

It is possible to use stateful session beans from the web container. However, a web container has a "lighter" means of managing application state—using an HttpSession object.

Singleton Session Beans

- A singleton session bean maintains state across method calls.
- State is shared by all clients because all clients will access the same bean instance.

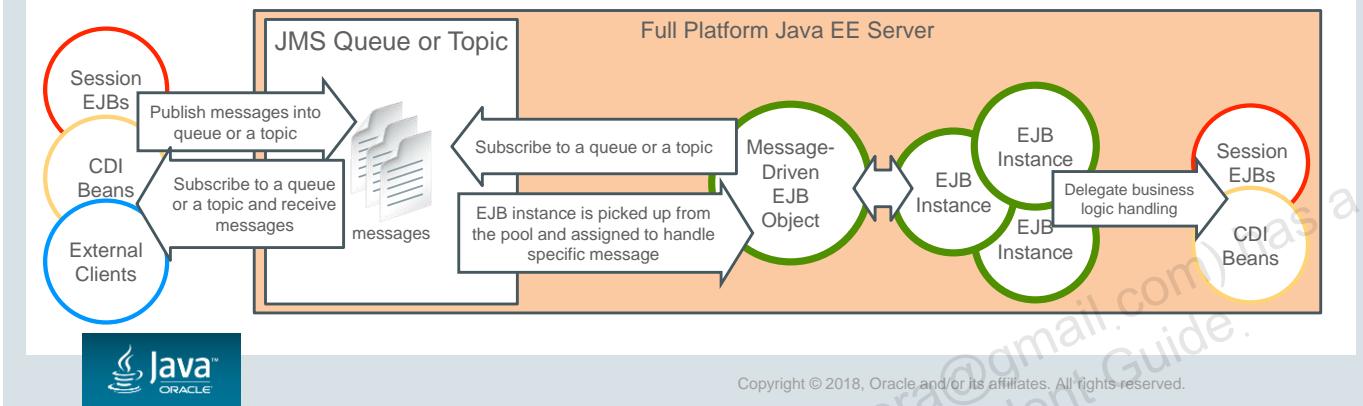
The methods of a singleton session bean can be concurrently accessed by multiple clients. By default, a singleton session bean's concurrent access is controlled by the container using Container Managed Concurrency. Container Managed Concurrency supports a multiple reader, single writer locking strategy. Every method of a singleton session bean attempts to obtain a write lock by default. Methods can be annotated with either of the following:

`@javax.ejb.Lock(javax.ejb.LockType.WRITE)`
`@javax.ejb.Lock(javax.ejb.LockType.READ)`

Message-Driven EJB

Message producers and consumers: Message-driven bean:

- Java EE components
- External clients
- Stateless asynchronous message consumer
- Can be subscribed to receive messages from queues or topics
- Can never be directly invoked by a client
- Typically is responsible for acquiring, validating, and preparing messages before passing on the business logic handling classes



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Java Message Service is a Java API for sending and receiving messages.

JMS Server is a part of the Full Java EE Platform specification. Its role is to act as an intermediary between Message Producers and Message Consumers, handling messages in-transit.

JMS Producer could be a non-Java client, a Java SE client, or a Java EE component, such as a session EJB, a CDI bean, and so on.

JMS Consumer could be a non-Java client, a Java SE client, or a Java EE component. There is a special type of Enterprise JavaBean—a Message-Driven Bean, which represents a Stateless Asynchronous Message Consumer in the full Java EE platform specification.

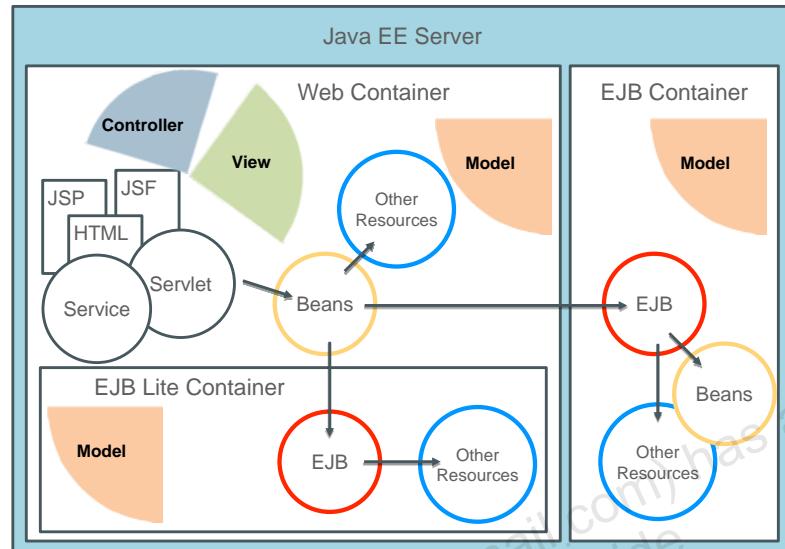
The JMS API in the Java EE platform has the following features:

- Application clients, Enterprise JavaBeans (EJB) components, and web components can send or synchronously receive a JMS message. Application clients can, in addition, set a message listener that allows JMS messages to be delivered to it asynchronously by being notified when a message is available.
- Message-driven beans, which are a kind of enterprise bean, enable the asynchronous consumption of messages in the EJB container. An application server typically pools message-driven beans to implement concurrent processing of messages.
- Message send and receive operations can participate in Java Transaction API (JTA) transactions, which allow JMS operations and database accesses to take place within a single transaction.

Assembling Application Components with CDIs

Context Dependency Injections and annotations are used to reference objects such as:

- POJOs
- JSF managed beans or CDI beans
- EJBs
- Other resources such as
 - EntityManager
 - JMS queues or topics
 - EJBContext
 - ServletContext
 - And so on



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

CDI managed beans and Enterprise JavaBeans are often used together. For example: Web Container Components use CDI managed beans that communicate with EJBs, or EJBs use CDI beans to reuse business logic implementation code. Components can also inject other resources such as EntityManager.

The choice between implementation types is life cycle–driven:

- Web container life cycles are determined by CDI and are linked to request handling. Typically this means that the Java objects that handle your application logic would be loaded in response to client calls.
- The EJB container life cycle is conditioned by a bean type. This means that the container can pre-load the required objects to memory and thus make them already available by the time they are called upon to handle business logic.

JSF Managed Beans, CDI Beans, EJBs

Enterprise JavaBeans (EJBs):

- Described by the javax.ejb package
- Can be used in the EJB Java EE container
- Can be called locally and remotely
- Can be stateful and be passivated
- Can work with timers
- Can be invoked asynchronously

CDI beans:

- Described by the javax.enterprise.context package
- Can be used in all Java EE Containers—not limited to JSF
- Support interceptors, events, and so on, and are more flexible than JSF Managed Beans
- In addition to annotations, may use beans.xml deployment descriptor

JSF managed beans:

- Described by the javax.faces.bean package
- Used in the web container by JSF Components
- CDI Beans "evolved" from JSF managed beans

❖ The following examples demonstrate the use of CDI beans in a context of a web container. However, they could also be used in EJB container.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

In Java EE 7, developers no longer had to explicitly add a beans.xml file in your application archive in order for CDI to work. However, it is still possible to use beans.xml file for more flexible control over CDI beans. This topic is covered in the Appendix B.

Request Scope

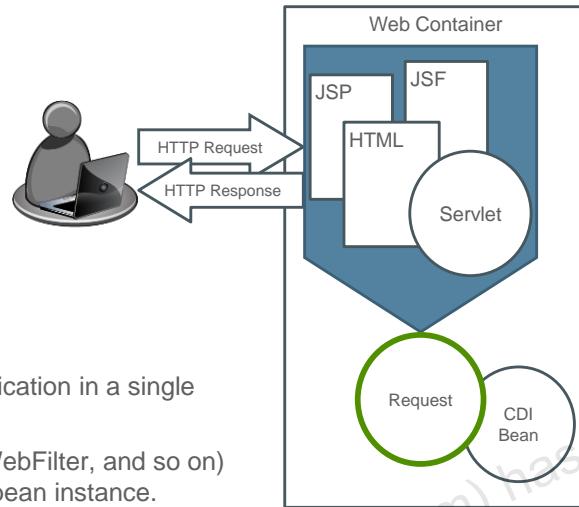
```
package demos;

import javax.enterprise.context.*;

@RequestScoped
public class MyBean {
    public void doSomething() {
        //...
    }
}
```

Request scoped beans:

- Instantiated once per user's interaction with a web application in a single HTTP request
- More than one server component (servlet, JSP, JSF, WebFilter, and so on) can handle the same client request, sharing the same bean instance.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Request Scope is defined by the `@RequestScoped` annotation.

It represents a user's interaction with a web application in a single HTTP request. A single request received from the client may be handled by a number of server-side components such as servlets, JSPs, JSFs, web services, and intercepting filters. No matter how many objects are used to perform this request handling they all share the same Request Scope.

Annotation used in this example:

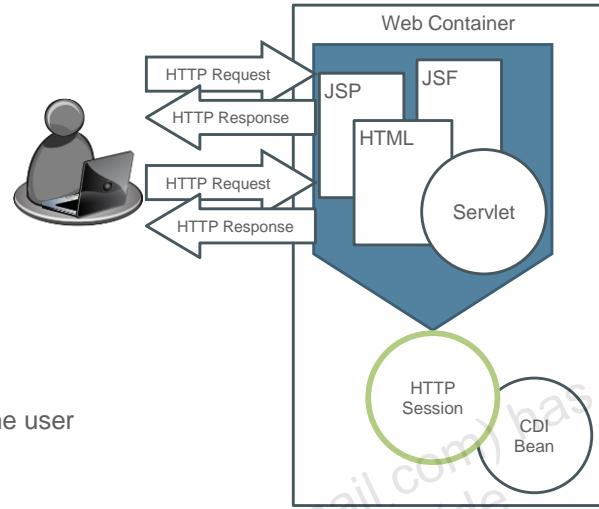
<http://docs.oracle.com/javaee/7/api/javax/enterprise/context/RequestScoped.html>

Session Scope

```
package demos;

import javax.enterprise.context.*;
import java.io.Serializable;

@SessionScoped
public class MyBean implements Serializable {
    public void doSomething()
    //...
}
```



Session Scoped Beans:

- Instantiated once per user session
- Shared across multiple HTTP requests from the same user



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Session Scope is defined by the `@SessionScoped` annotation.

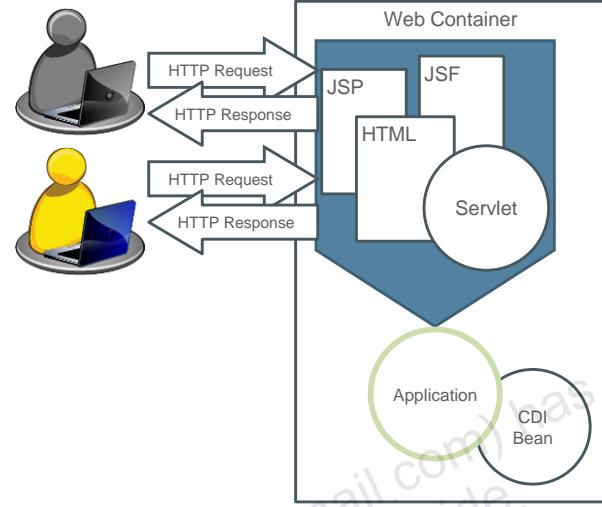
It represents a user's interaction with a web application across multiple HTTP requests. All servlets, JSPs, JSFs, and web services invoked by the same client (typically the same browser session) would be able to share same Session Scope.

Annotation used in this example:

<http://docs.oracle.com/javaee/7/api/javax/enterprise/context/SessionScoped.html>

Application Scope

```
package demos;  
  
import javax.enterprise.context.*;  
  
@ApplicationScoped  
public class MyBean {  
    public void doSomething() {  
        //...  
    }  
}
```



Application Scoped Beans:

- Instantiated once per application
- Shared by all application users



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Application Scope is defined by the `@ApplicationScoped` annotation.

It represents a shared state across all users' interactions with a web application.

Annotation used in this example:

<http://docs.oracle.com/javaee/7/api/javax/enterprise/context/ApplicationScoped.html>

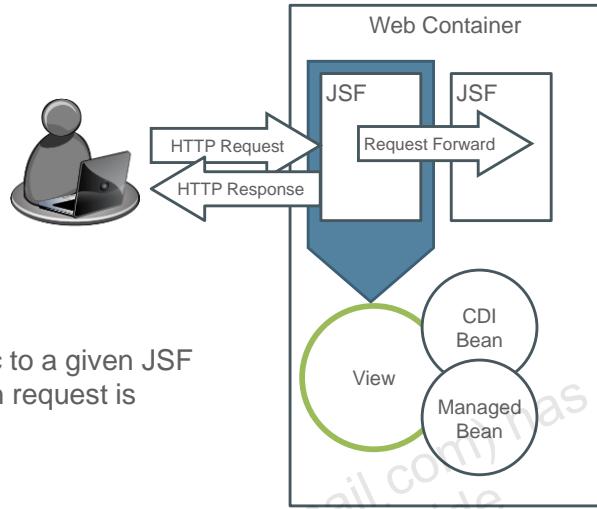
View Scope

```
package demos;
import javax.inject.*;
import javax.faces.view.*;
@Named("aBean")
@ViewScoped
public class MyCDIBean { ... }
```

OR

```
package demos;
import javax.faces.bean.*;
@ManagedBean("bBean")
@ViewScoped
public class MyManagedBean { ... }
```

View Scoped is a JSF-specific scope that is specific to a given JSF page. It is not "shared" with another JSF page when request is forwarded to it during navigation.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

View Scope is defined by the `@ViewScoped` annotation. It is used by JSF components to introduce a scope similar to the session scope, but limited to just one JSF page, in case there is a forward action from this page to another.

Although, View Scope is JSF-specific, in Java EE 7 you can enable not only JSF Managed Beans, but also CDI Beans for this scope.

To create an alias that JSF page can reference JSF Managed Beans use `@ManagedBean` annotation and CDI beans can use `@Named` annotation instead

Annotations used in this example:

<http://docs.oracle.com/javaee/7/api/javax/faces/view/ViewScoped.html>

<http://docs.oracle.com/javaee/7/api/javax/inject/Named.html>

<http://docs.oracle.com/javaee/7/api/javax/annotation/ManagedBean.html>

<http://docs.oracle.com/javaee/7/api/javax/faces/bean/ViewScoped.html>

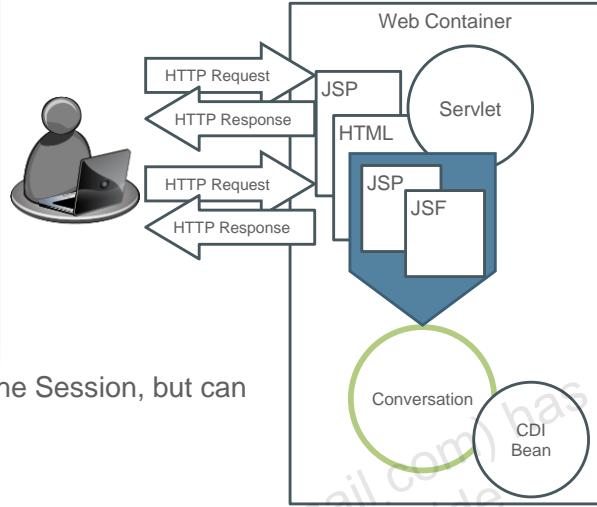
Conversation Scope

```
package demos;

import javax.enterprise.context.*;
import java.io.Serializable;

@ConversationScoped
public class MyBean implements Serializable {
    @Inject
    private Conversation conversation;
    public void startConversation() {
        conversation.begin();
    }
    public void endConversation() {
        conversation.end();
    }
}
```

Conversation Scoped is a custom scope similar to the Session, but can be limited to specific part of application.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

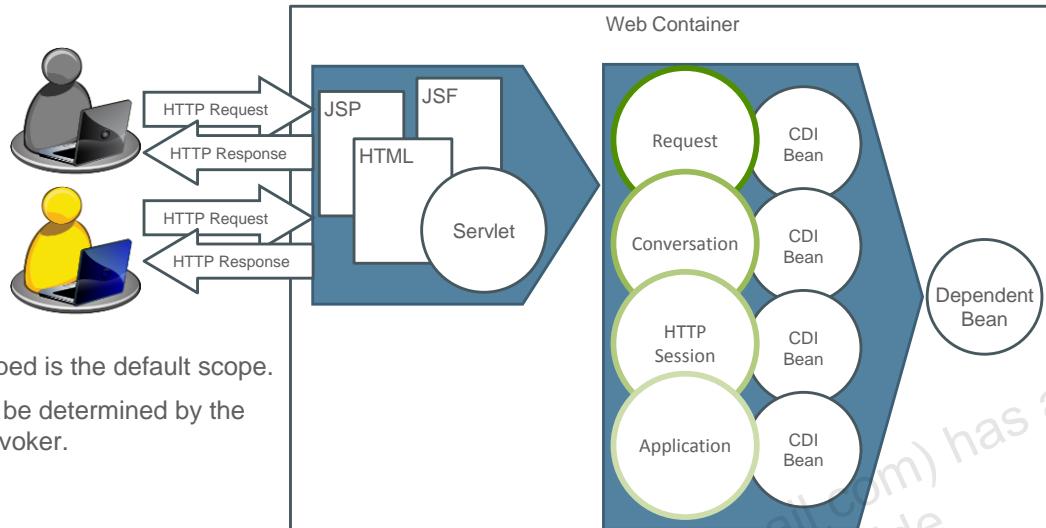
Conversation Scope is defined by the `@ConversationScoped` annotation. It represents a user's interaction with servlets, JSPs, JSFs, and web services application components. The conversation scope exists within developer-controlled boundaries that extend it across multiple requests for long-running conversations.

If your conversation should be made visible to all web components and last for the duration of the user session, you may simply use Session Scope instead.

Annotation used in this example:

<http://docs.oracle.com/javaee/7/api/javax/enterprise/context/ConversationScoped.html>

Dependent Scope



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Dependent Scope is defined by the @Dependent annotation.

It is a default scope if none is specified; it means that an object exists to serve exactly one client (bean) and has the same lifecycle as that client (bean).

Annotation used in this example (this is a default, so actual annotation had to be present):

<http://docs.oracle.com/javaee/7/api/javax/enterprise/context/Dependent.html>

Injecting Beans

ProductOrder bean definition with "Product" property, "placeOrder" operation, and optional alias "order":

```
package demos;
@Named("order")
@RequestScoped
public class ProductOrder {
    private String productName;
    public String getProduct() {
        return productName;
    }
    public void setProduct(String name) {
        productName = name;
    }
    public void placeOrder() {
        //...
    }
}
```



Injection of the ProductOrder bean into JSF page using "order" alias:

```
<h:form>
<h:inputText id="name" value="#{order.product}" />
<h:commandButton value="Ok" action="#{order.placeOrder}" />
</h:form>
```

Injection of the ProductOrder bean into OrderManagement class:

```
package demos;
public class OrderManagement {
    @Inject;
    private ProductOrder productOrder;
    public void handleOrder() {
        String name = productOrder.getProduct();
        productOrder.placeOrder();
    }
}
```

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

To inject a CDI bean you can simply use `@Inject` annotation associated with a variable of the corresponding bean type.

`@Named` Annotation can be used as a qualifier to specify the name of an implementation that needs to be injected

```
public interface Order {...}
@Named("PO")
@RequestScoped
public class ProductOrder implements Order {...}
@Named("SO")
@RequestScoped
public class ServiceOrder implements Order {...}
public class OrderManagement {
    @Inject;
    private @Named("PO") Order order;
    ...
}
```

`@Named` Annotation is also used to give a CDI bean an alias that can be referenced by the JSP or JSF page: <http://docs.oracle.com/javaee/7/api/javax/inject/Named.html>

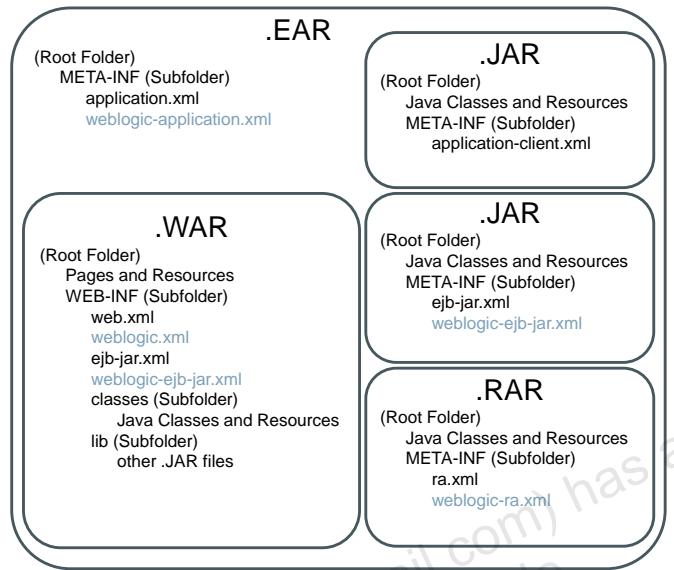
JSF managed beans define alias with the `ManagedBean` annotation:

<http://docs.oracle.com/javaee/7/api/javax/annotation/ManagedBean.html>

Java EE Packaging and Deployment

JSR 88: Java EE Application Deployment

- Enterprise Archive
 - Packaged as EAR files
 - Contains other modules
- Web Module - WAR files
 - Packaged as WAR files
 - Contains web content such as HTML, servlet, JSP, JSF etc..
 - May contain library or EJB JAR files
- EJB Module
 - Packaged as JAR files
 - Contains Enterprise JavaBeans
- Resource Adapter Module
 - Packaged as RAR files
 - Contains JCA adapters
- Application Client Module
 - Packaged as JAR files
 - Contains Java client applications



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

JSR 88: Java TM EE Application Deployment specification provides a complete description of the APIs required by the J2EE platform to enable development of platform-independent deployment tools.

<https://jcp.org/en/jsr/detail?id=88>

Java EE applications are delivered in a Java Archive (JAR) file, a Web Archive (WAR) file, or an Enterprise Archive (EAR) file. A WAR or EAR file is a standard JAR (.jar) file with a .war or .ear extension.

Optionally these archives may contain deployment descriptors XML documents, that describe the deployment settings of an application, a module, or a component. Because deployment descriptor information is declarative, it can be changed without the need to modify the source code. At runtime, the Java EE server reads deployment descriptors and acts upon the application, module, or component accordingly.

Instead of or in addition to deployment descriptor XML files, developer can specify deployment information using annotation placed in the source code. Deployment descriptors, if present, override what is specified in the source code.

There are two type of deployment descriptor files:

- Java EE – These are generic and are described by the JSR88 standard. They are: application.xml, web.xml, ejb-jar.xml, application-client.xml , ra.xml.
- Runtime – These are specific to a target Java EE Server provider. For example for a WebLogic server, they would be: weblogic-application.xml, weblogic.xml, weblogic-ejb-jar.xml, weblogic-ra.xml

Annotations or Deployment Descriptors

In Java EE 7, deployment descriptors are optional - a developer may use annotations instead. This example shows an EJB component described both ways.

Using Annotations:

```
package demos;

@Stateless(name="Orders")
public class OrderEntry {
    public void placeOrder() {
        //...
    }
}
```

These two options are available for all Java EE components.

Using ejb-jar.xml Deployment Descriptor:

```
<ejb-jar xmlns="http://xmlns.jcp.org/xml/ns/javaee"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
          http://xmlns.jcp.org/xml/ns/javaee/ejb-jar_3_2.xsd"
          version="3.2">
    <enterprise-beans>
        <session>
            <ejb-name>Orders</ejb-name>
            <ejb-class>demos.OrderEntry</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Container</transaction-type>
        </session>
    </enterprise-beans>
</ejb-jar>
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Annotations are significantly simpler to use. They allow developers to directly describe their code, without having to create XML deployment descriptor files every time. However, this approach is not without its limitations. Annotations appear to be less flexible than XML deployment descriptors, because their modifications require developers to modify actual Java classes, rather than separate XML files. The next page shows that annotations can be used together with XML descriptors to achieve better code management flexibility.

In some cases configuration can only be achieved by using deployment descriptor files. For example, application.xml file is used to set the web context root binding for the web module:

```
<application xmlns="http://java.sun.com/xml/ns/j2ee" version="1.4"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/
http://java.sun.com/xml/ns/j2ee/application_1_4.xsd">
    <display-name>ProductApp</display-name>
    <module>
        <ejb>product-ejb.jar</ejb>
    </module>
    <module>
        <web>
            <web-uri>web-ui.war</web-uri>
            <context-root>products</context-root>
        </web>
    </module>
</application>
```

Annotations with Deployment Descriptors

Deployment descriptors and annotation can be used together.

- Annotations are convenient.
- Descriptors are flexible.
- Properties and behaviors of all Java EE components can be adjusted via XML descriptors without changing annotations in the source code.

```
package demos;

@Stateless(name="Orders")
public class OrderEntry {
    @Resource(name="mailhost")
    String mailServer;
    public void placeOrder() {
        //...
    }
}
```

```
<ejb-jar xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
    http://xmlns.jcp.org/xml/ns/javaee/ejb-jar_3_2.xsd"
    version="3.2">
    <enterprise-beans>
        <session>
            <ejb-name>Orders</ejb-name>
            <ejb-class>demos.OrderEntry</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Container</transaction-type>
            <env-entry>
                <description>EMail server host</description>
                <env-entry-name>mailhost</env-entry-name>
                <env-entry-type>java.lang.String</env-entry-type>
                <env-entry-value>smtp.example.com</env-entry-value>
            </env-entry>
        </session>
    </enterprise-beans>
</ejb-jar>
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

In this example, an ejb-jar.xml descriptor file is used to create an environment entry that can be altered at deployment time. Such entries can be accessed from your Java code via JNDI lookups or annotations. This approach can be used for a variety of different types of Objects and not just for environment entries, but lots of other properties and behaviors of your classes that you may wish to adjust without having to change the actual class source code. Use deployment descriptors for properties and behaviors that can potentially need configuration without the need to modify, or have access to, the Java source code.

Java Naming Directory Interface Objects

JNDI is used to catalog various types of objects, such as:

- EJBs
- JMS queues, topics, connection factories
- Data sources
- LDAP objects
- etc...

```
@Stateless
public class OrderManagement { ... }
```

```
@DataSourceDefinition (
    name="java:global/ProductsApp/productDB",
    className="org.apache.derby.jdbc.ClientDataSource",
    url="jdbc:derby://localhost:1527/ProductDB",
    user="...",
    password="...",
    databaseName="ProductDB"
)
```

Global JNDI naming convention:

```
java:global/<application-name>/<module-name>/<bean-name>
      java:app/<module-name>/<bean-name>
        java:module/<bean-name>
          <bean-name>
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

JNDI is an API specified in Java technology that provides naming and directory functionality to applications written in the Java programming language. It is designed especially for the Java platform using Java's object model. Using JNDI, applications based on Java technology can store and retrieve named Java objects of any type. In addition, JNDI provides methods for performing standard directory operations, such as associating attributes with objects and searching for objects using their attributes.

JNDI is also defined independently of any specific naming or directory service implementation. It enables applications to access different, possibly multiple, naming and directory services using a common API. Different naming and directory service providers can be plugged in seamlessly behind this common API. This enables Java technology-based applications to take advantage of information in a variety of existing naming and directory services, such as LDAP, NDS, DNS, and NIS(YP), as well as enabling the applications to coexist with legacy software and systems.

Using JNDI as a tool, you can build new powerful and portable applications that not only take advantage of Java's object model but are also well-integrated with the environment in which they are deployed.

Portable Global JNDI name in EJB 3.1

- EJB 3.1 solves the above problem by mandating that every container must assign (at least one) well defined global JNDI names to EJBs.
- <application-name> defaults to the bundle name (.ear file name) without the bundle extension. This can be overridden in application.xml. Also, <application-name> is applicable only if the bean is packaged inside an .ear file.

- <module-name> defaults to bundle name (.war or .jar) without the bundle extension. Again, this can be overridden in ejb-jar.xml.
- <bean-name> defaults to the unqualified class name of the bean. However, if @Stateful or @Stateless or @Singleton uses the name attribute, then the value specified there will be used as the bean name.

When referencing an EJB that has remote or local interfaces, you can look up a specific interface by using this syntax:

```
java:global/<application-name>/<module-name>/<bean-name>!<fully-qualified-bean-interface-name>
```

Resources could be described as annotations, as in the example with the data source or via xml configuration files:

```
<?xml version="1.0" encoding="UTF-8"?>
<jdbc-data-source>
    <name>ProductDB</name>
    <jdbc-driver-params>
        <url>...</url>
        <driver-name>...</driver-name>
        <properties>...</properties>
    </jdbc-driver-params>
    <jdbc-data-source-params>
        <jndi-name>jdbc/productDB</jndi-name>
    </jdbc-data-source-params>
</jdbc-data-source>
```

Container-Managed Injections

Components in Java EE environment use annotations to inject resources and dependencies.

Injecting Resources:

- Data sources
- JMS queues, topics
- etc...

```
@Resource(lookup="java:global/ProductsApp/productDB")
private DataSource myDB;
```

Defining Components:

- CDI beans
- Enterprise JavaBeans

```
@RequestScoped
public class ProductOrder {...}
```

```
@Stateless
public class OrderManagement {...}
```

Injecting Dependencies:

- CDI beans
- Enterprise JavaBeans

```
@Inject
private ProductOrder po;
```

```
@EJB
private OrderManagement om;
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Resource injection allows injection of any resource available in the JNDI namespace into any container-managed object, such as a servlet, an enterprise bean, or a managed bean. Dependency injection allows regular Java classes to be turned into managed objects (CDI beans) and to inject them into any other objects managed by the container.

Optionally when looking up an EJB component, you can use its global JNDI name:

```
@EJB (lookup="<jndi-name>")
private OrderManagement om;
```

If a bean in one scope tries to refine another bean in different scope, it would be able to do so via the container-managed proxy. This means that if the RequestScoped bean tries to access SessionScoped or ApplicationScoped beans, it can do it directly, because only one session or application context exists for any given request at a time. However, if a SessionScoped bean tries to access bean that is RequestScoped, a proxy would be required to resolve a reference to a current request, because there could be many requests linked to a single session. In Java EE 7 such proxies are provided automatically.

- Resource injection can inject JNDI resources directly.
- Dependency injection can inject regular classes directly.
- Resource injection resolves by resource name, whereas dependency injection resolves by type.

Multiple resources can be defined together using `@javax.annotation.Resources` annotation

JNDI Lookups

Components outside of the Java EE container environment perform explicit JNDI lookups.

Create Initial Context Object to reference JNDI:

- Using Java code to set server context properties:
- Or using the `jndi.properties` file:

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
        "weblogic.jndi.WLInitialContextFactory");
env.put(Context.PROVIDER_URL,
        "t3://localhost:7001");
Context context = new InitialContext(env);
```

```
java.naming.factory.initial=weblogic.jndi.WLInitialContextFactory
java.naming.provider.url=t3://localhost:7001
```

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

Perform lookups using JNDI object names:

```
DataSource ds = (DataSource)context.lookup("<data-source-name>");
OrderManager om = (OrderManager)context.lookup("<ejb-name>");
```

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Summary

In this lesson, you studied the fundamentals of Java EE 7 Architecture and its components. After completing this lesson, you should have learned how to describe:

- Standards, containers, APIs, and services
- Application component functionalities mapped to tiers and containers
 - Web container technologies
 - Business logic implementation technologies
 - Web service technologies
- Packaging and deployment
- Enterprise JavaBeans, managed beans, and CDI beans
 - Understanding lifecycle and memory scopes
- Linking components together with annotations, injections, and JNDI



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



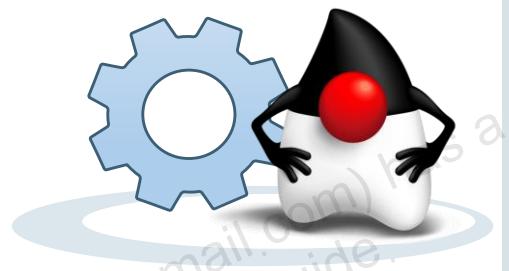
Practices

In this practice you configure your environment to support Java EE 7 application development

- Configure and start WebLogic Server instance
- Configure a Derby Database and populate it with data



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Jairo Jose Silvera Sarmiento (jairo.silvera@gmail.com) has a
non-transferable license to use this Student Guide.

Managing Persistence by Using JPA Entities

3



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



ORACLE®

Jairo Jose Silvera Sarmiento (jairo.silvera@gmail.com) has a
non-transferable license to use this Student Guide.

Objectives

This lesson describes management of persistence by using the Java Persistence API. After completing this lesson, you should be able to:

- Create JPA entities with Object-Relational Mappings (ORM)
- Use Entity Manager to perform database operations with JPA entities
- Handle entity data with conversions, validations, and key generation
- Describe persistence management and locking mechanisms
- Create and execute JPQL statements



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

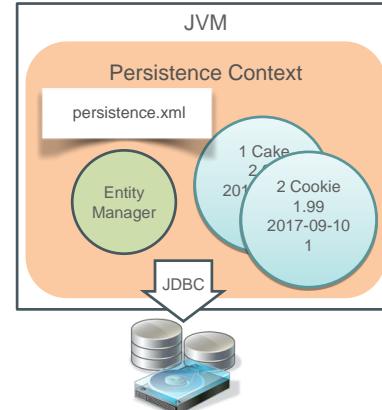


Java Persistence API

JPA is a lightweight framework that is designed to represent relational database data as POJOs.

JPA is built on top of JDBC and addresses the complexity of managing both SQL and Java code.

- JPA is designed to facilitate object-relational mapping.
- JPA works in both Java SE and Java EE environments.
- ORM Providers (EclipseLink, Hibernate) implement JPA operations.
- The key JPA concepts include:
 - Entities (POJOs mapped to database objects)
 - Persistence units (configuration described in `persistence.xml`)
 - Persistence contexts (In-Memory set of Entity instances)
- EntityManager is used to perform operations on entities:
 - `find`
 - `persist`
 - `merge`
 - `remove`



| PRODUCTS | | | | |
|----------|--------|-------|-------------|---------|
| id | name | price | best_before | version |
| 1 | Cake | 2.99 | 2017-10-10 | 1 |
| 2 | Cookie | 1.99 | 2017-09-10 | 1 |

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



The Java Persistence API (1.0) began as a part of the Enterprise JavaBeans 3.0 specification (JSR-220) to standardize a model from object-relational mapping.

JPA 2.0 (JSR-317) set out to improve on the original JPA specification and was defined in its own JSR. JPA is currently at version 2.1 (JSR-338).

JDBC was the first mechanism that Java developers used for persistent storage of data. However, working with JDBC requires that you understand how to map a Java object to a database table and maintain the set of SQL queries that is used to transform relational data into Java objects. JPA provides a framework for this object-relational mapping while preserving the ability to manipulate databases directly.

The key JPA concepts in this lesson are the starting point for writing JPA applications:

- An entity can be used to represent a relational table by a Java object. (This is a one-to-one mapping.)
- A persistence unit defines the set of all classes that are related or grouped by the application and that must be collocated in their mapping to a single database.
- A persistence context is a set of entity instances in which there is a unique entity instance for any persistent entity identity.

The storage mechanism that is most often used by applications is a relational database. A relational database is typically configured to store data in tables that have a relationship with one another. However, an object-oriented application design may not have the same organization in the same structure. A Java domain object can encompass partial data from a single database table, or it can include data from multiple tables depending on the normalization of the relational database.

Persistence mechanisms are the code that enables programmers to persist (store) data in a relational database. JDBC and JPA are two examples of persistence mechanisms.

Writing code to translate a relational schema to an object-oriented domain schema (or vice versa) can be time consuming and error prone. The object-relational mapping (ORM) software frameworks attempt to manage the mapping to object-oriented programmers while requiring little coding. EclipseLink and Hibernate are examples of ORM software.

In the diagram in the slide, on the left is the most basic ORM mapping: a simple mapping of Java objects to database tables (one-to-one mapping).

JPA Entities: Basics

```
package demos.db;
import javax.persistence.*;

@Entity
@Table(name="PRODUCTS")
public class Product implements Serializable {
    @Id
    @NotNull
    private Integer id;
    @NotNull
    private String name;
    @NotNull
    private BigDecimal price;
    @NotNull
    @Temporal(TemporalType.DATE)
    @Column(name="best_before")
    private LocalDate bestBefore;
    @Version
    @NotNull
    private Integer version;
    /* constructors, get, set,
     * equals, hashCode etc. operations */
}
```



JPA Entities:

- **Classes mapped to Database Tables or Views**
- Attributes mapped to Columns
- **Unique Identity** must be specified.
- **Validations and Constraints** may be applied.
- **Temporal** attributes represent Date, Time, or Timestamp database types.
- **Names may be overridden if required.**
- **Version** attribute may be used to handle locking.
- **Entity instances** represent records.



| PRODUCTS | | | | |
|----------|--------|-------|-------------|---------|
| id | name | price | best_before | version |
| 1 | Cake | 2.99 | 2017-10-10 | 1 |
| 2 | Cookie | 1.99 | 2017-09-10 | 1 |

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

A JPA entity is a Java object that represents something to be persisted—a Customer, an Employee, a Book, and so on. Basically, anything that is typically represented in a relational database can be represented by a POJO that defines the elements of the entity and includes methods to set and get the values of each element.

The JPA characteristics of entities include:

- Persistence: Entities that are created can be stored in a database (persisted) yet treated as objects.
- Identity: Each entity has a unique object identity, which is generally associated with a key in the persistent store (database). Typically, the key is the database primary key.
- Transactions: JPA requires a transactional context for operations that commit changes to the database.
- Granularity: JPA entities can be as fine-grained or coarse-grained as a developer wants, typically representing a single row in a table.

Entities are managed by the EntityManager API.

- An entity can be created from a POJO through annotations.
- The entity class must be annotated with the `@Entity` annotation.
- The entity class must have a no-arg constructor. The no-arg constructor must be public or protected. Other constructors can still be used.
- The entity class must be a top-level class but can be a concrete or an abstract class.
- The persistent state of an entity is represented by instance fields.
- Entities support inheritance and must not be final.

- If an entity instance is to be passed by value as a detached object (for example, through a remote interface), the entity class must implement the `java.io.Serializable` interface.
- Note that instance fields must be accessed only from within the methods of the entity by the entity instance itself; they must be declared private.
- An entity cannot be an enum or an interface.
- No methods or persistent instance variables of the entity class may be final.
- Given the definition of an entity class and what you know about CDI beans, it might be tempting to simply inject an entity as a managed bean into another managed bean. This is discouraged by the designers of CDI because JPA cannot persist injected CDI proxies.

In the example in the slide, you can see the `@Entity` annotation to declare the `Product` class as an entity class to manage. Because the name of the entity class differs from the name of the table, a `@Table` annotation is used to map these object names.

The `@Id` annotation declares the primary key.

The field names match the table column names (attributes), except the `bestBefore` field, which is mapped to a database column with the help of the `@Column` annotation to match the column name.

The `@Temporal` annotation is required for any persisted field that is of the `java.util.Date` or `java.util.Calendar` type. Because databases store dates in different ways, some as `TimeStamp` types and some as `Date` types, the `@Temporal` annotation indicates that you want Java to manage the conversion to and from the Java Date or Calendar type to the appropriate type in the database.

Alternatively to Annotations, Entities can also be mapped to database object with the `orm.xml` file.

The `orm.xml` file is typically specified in the `META-INF` directory in the root of the persistence unit, or in the `META-INF` directory of any JAR file that is referenced by `persistence.xml`. Additionally, one or more mapping files can be referenced by the `mapping-file` elements of the `persistence-unit` element. These mapping files can be present anywhere on the class path.

Note: If you give the object-relational mapping XML file a name other than `orm.xml`, you must include the `mapping-file` element in `persistence.xml`.

```
<persistence>
    <persistence-unit name="ProductPU">
        <mapping-file>customMappings.xml</mapping-file>
    </persistence-unit>
</persistence>
```

This is an example content of the object relational mappings file:

```
<entity-mapping ...>
    <entity class="demos.db.Product">
        <attributes>
            <id name="id">
                <column name="ID" />
            </id>
            <basic name="name" />
            <basic name="price" />
        </attributes>
    </entity>
</entity-mapping>
```

Persistent Field Versus Persistent Property

Using Property instead of the Field annotation allows you to perform extra processing of value.

- Properties are annotated against "get" operations.
- Fields are annotated against attributes.
- If a field has the same name as a column, no annotations are required (default).
- Transient fields are not persisted.

```
Persistent Field "price"
@Entity
public class Product implements Serializable {
    ...
    private BigDecimal price;
    public BigDecimal getPrice() {
        return price;
    }
    public void setPrice(BigDecimal price) {
        this.price = price;
    }
}
```

```
Persistent Property "price"
Transient Field "discount"
@Entity
public class Product implements Serializable {
    ...
    private BigDecimal price;
    @Transient
    private BigDecimal discount;
    @Column(name="price")
    public BigDecimal getPrice() {
        return price.subtract(discount);
    }
    public void setPrice(BigDecimal price) {
        this.price = price;
        this.discount =
            price.multiply(BigDecimal.valueOf(0.1));
    }
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Entity fields:

- They cannot be public.
- They should not be read by clients directly.
- Unless they are annotated with the `@Transient` annotation or the `transient` keyword, all fields are persisted. (The use of the `@Column` annotation defines only the column name to use.)

When fields are annotated, the persistence provider uses reflection to get and set the fields of the entity.

Note: Accessor methods can be present in an entity class that uses field-based access and may be useful for other types of operations, but they are ignored by the persistence provider.

When using persistent properties, the persistence provider will retrieve an object's state by calling the accessor methods of the entity class.

- Methods must be declared as public or protected.
- Methods must follow the JavaBeans naming convention.
- `@Column` may also be used to define the column name in the database.
- Persistence annotations can be placed only on getter methods.

The choice between property and field is really about whether you want to do any additional processing in a getter/setter method.

Using Access Annotation

Access Annotation overrides Property and Field Access Types.

In this example:

- At class level, access type is set to FIELD.
- All fields are persisted as usual.
- Field price changes its access type handling to PROPERTY in order to enable additional processing in its get/set operations.
- Field price is marked as Transient (not persisted as other fields) to avoid duplication with the getPrice operation marked with PROPERTY access type.
- Filed discount is Transient and will not be persisted.

```
@Entity
@Access (AccessType.FIELD)
public class Product implements Serializable {
    @Id
    private Integer id;
    @Transient
    private BigDecimal discount;
    @Transient
    private BigDecimal price;

    @Access (AccessType.PROPERTY)
    public BigDecimal getPrice() {
        return price.subtract(discount);
    }
    public void setPrice(BigDecimal price) {
        this.price = price;
        this.discount =
            price.multiply(BigDecimal.valueOf(0.1));
    }
    ...
}
```

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



To use both Field and Property access types for the same entity, you should add a class-level @Access annotation to define the default access type for the entity class and additional @Access annotations to the individual persistent fields or properties that should be different from the default access type.

- If the class is designated with field access, annotate the property (getter methods) with @Access(AccessType.PROPERTY).
- If the class is designated with property access, annotate the fields with @Access(AccessType.FIELD).

Note: When different access types are combined within the same class, the @Transient annotation should be used to avoid duplicate mappings.

Converters

A converter defines an alternative type of transformation.

- Converters can be applied **globally**, to all fields of a particular type or
- A specific entity field can directly **reference a converter**
- @Convert(disableConversion=true)** disables global automatic conversion for a specific field.

```
package demos.db;
@Entity
public class Product implements Serializable {
    ...
    @Temporal(TemporalType.DATE)
    @Column(name="best_before")
    @Convert(converter=DateConverter.class)
    private LocalDate bestBefore;
}
```

```
@Converter(autoApply=true)
public class DateConverter implements AttributeConverter<LocalDate, Date> {
    @Override
    public Date convertToDatabaseColumn(LocalDate value) {
        return (value==null) ? null :
            Date.from(value.atStartOfDay(ZoneId.systemDefault()).toInstant());
    }
    @Override
    public LocalDate convertToEntityAttribute(Date value) {
        return (value==null) ? null :
            Instant.ofEpochMilli(value.getTime()).atZone(ZoneId.systemDefault()).toLocalDate();
    }
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

When the optional autoApply property is set to "true", the converter will be applied to all attributes of a specific type—in the example above, it is LocalDate.

To disable automatic conversion for a particular attribute, the **@Convert(disableConversion=true)** annotation should be applied to the attribute.

Persistence fields or properties can be of the following data types:

Java primitive types and Java wrappers, including:

- Integer, Boolean, Float, and Double

Serializable types, including:

- String, BigDecimal, and BigInteger

Arrays of bytes and characters, including:

- byte[] and Byte[], char[] and Character[]

Temporal types, including (but not limited to):

- java.util.Date, java.util.Calendar, java.sql.Date, and java.sql.TimeStamp

Enums and collections

When implementing relationship mappings, another entity, or a collection of entities becomes an attribute type.

In this example a LocalDate class is used to represent the bestBefore Date item. Depending on the version of the JPA provider, this may require additional conversion. The JPA 2.1 (latest) specification is not part of Java EE, but the Java SE product set, and has been created before Java SE 8.

Therefore, unless a JPA provider specifically supports Java SE 8 features, it would not be able to directly handle Java SE 8 specific types such as LocalDate, LocalTime, or LocalDateTime.

Generated Keys

Primary Key column values can be generated by using database sequences or tables.

SequenceGenerator

```
import javax.persistence.*;
@Entity
public class Product implements Serializable {
    @Id
    @NotNull
    @GeneratedValue(strategy=SEQUENCE,
                    generator="seq_idGen")
    @SequenceGenerator(name="seq_idGen",
                       sequenceName="PID_SEQ")
    private Integer id;
    ...
}
```

TableGenerator

```
import javax.persistence.*;
@Entity
public class Product implements Serializable {
    @Id
    @NotNull
    @GeneratedValue(strategy=TABLE,
                    generator="tab_idGen")
    @TableGenerator(name="tab_idGen",
                   table="ID_GEN",
                   pkColumnName="GEN_KEY",
                   valueColumnName="GEN_VALUE",
                   pkColumnValue="PID_SEQ")
    private Integer id;
    ...
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The GeneratedValue annotation may be applied to a primary key property or field of an entity or mapped superclass in conjunction with the Id annotation. The use of the GeneratedValue annotation is required to be supported only for simple primary keys. Use of the GeneratedValue annotation is not supported for derived primary keys.

The Generated Value annotation has following properties:

- generator (Optional): The name of the primary key generator to use as specified in the SequenceGenerator or TableGenerator annotation
- strategy (Optional): The primary key generation strategy that the persistence provider must use to generate the annotated entity primary key. This could be SequenceGenerator or TableGenerator.

A Sequence Generator defines a primary key generator that may be referenced by name when a generator element is specified for the GeneratedValue annotation. A sequence generator may be specified on the entity class or on the primary key field or property. The scope of the generator name is global to the persistence unit (across all generator types).

The SequenceGenerator annotation has following properties:

- allocationSize (Optional): The amount to increment by when allocating sequence numbers from the sequence
- catalog (Optional): The catalog of the sequence generator
- initialValue (Optional): The value from which the sequence object is to start generating
- schema (Optional): The schema of the sequence generator
- sequenceName (Optional): The name of the database sequence object from which to obtain primary key values

Defines a primary key generator that may be referenced by name when a generator element is specified for the GeneratedValue annotation. A table generator may be specified on the entity class or on the primary key field or property. The scope of the generator name is global to the persistence unit (across all generator types).

The TableGenerator annotation has the following properties:

- allocationSize (Optional): The amount to increment by when allocating ID numbers from the generator
- catalog (Optional): The catalog of the table
- indexes (Optional): Indexes for the table
- initialValue (Optional): The initial value to be used to initialize the column that stores the last value generated
- pkColumnName (Optional): Name of the primary key column in the table
- pkColumnValue (Optional): The primary key value in the generator table that distinguishes this set of generated values from others that may be stored in the table
- schema (Optional): The schema of the table
- table (Optional): Name of the table that stores the generated ID values
- uniqueConstraints (Optional): Unique constraints that are to be placed on the table
- valueColumnName (Optional): Name of the column that stores the last value generated

JPA Lifecycle Callback Methods

JPA Lifecycle Callback Methods are invoked by the EntityManager when it handles the Entity instance.

- Pre and Post Persist events are triggered around the insert of the new record.
- Pre and Post Update events are triggered around the update of the existing record.
- Pre and Post Remove events are triggered around the delete of the existing record.
- The PostLoad method is invoked when an entity is loaded into the current persistence context from the database or after the refresh operation is applied to it.

In these operations, you can:

- Validate Data
- Handle Data Denormalizations
- Set defaults and calculate values and so on

```
@Entity
public class Product implements Serializable {
    @Id
    @NotNull
    private Integer id;
    @NotNull
    private String name;
    @PrePersist
    protected void beforeInsert() {...}
    @PostPersist
    protected void afterInsert() {...}
    @PreUpdate
    protected void beforeUpdate() {...}
    @PostUpdate
    protected void afterUpdate() {...}
    @PreRemove
    protected void beforeDelete() {...}
    @PostRemove
    protected void afterDelete() {...}
    @PostLoad
    protected void afterSelectOrRefresh() {...}
    ...
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

An entity has a life cycle depending on the operations performed on that entity. Life-cycle callbacks are annotations that can be applied to entity methods that are invoked when an event occurs.

For entities to which a merge operation has been applied and causes the creation of newly managed instances, the `PrePersist` callback method is invoked for the managed instance after the entity state has been copied to it. These `PrePersist` and `PreRemove` callbacks are also invoked on all entities to which the operations are cascaded. The `PrePersist` and `PreRemove` methods are always invoked as part of the synchronous persist, merge, and remove operations.

The following rules apply to life-cycle callbacks:

- The callback methods can have public, private, protected, or package level access, but must not be static or final.
- A single method may be annotated with multiple life-cycle events, but only one callback method per life-cycle event should be used in a given entity. For example, an entity cannot have two different callback methods annotated with `@PostLoad`.
- Life-cycle callback methods may throw unchecked or runtime exceptions. A runtime exception thrown by a callback method that executes in a transaction causes that transaction to be marked for rollback.
- Lifecycle callbacks can invoke JNDI, JDBC, JMS, and enterprise beans. The life-cycle method of a portable application should not invoke EntityManager or Query operations, access other entity instances, or modify relationships in the same persistence context.
- A life-cycle callback method can modify the nonrelationship state of the entity on which it is invoked.

Validating Entities

Bean Validation 1.1 (JSR-349) enables developers to use annotations to apply constraints to types, methods, parameters, fields, or other annotations in JPA, JSF, CDI, and JAX-RS APIs.

- JPA controls application of Bean Validation via the `persistence.xml` file.
- Validation Mode can be used to switch validations on and off.
 - NONE: Turn off validation.
 - AUTO: Turn on validation when there is a validation provider on the classpath (the default).
 - CALLBACK: Turn on validation. The persistence provider throws a `PersistenceException` if the validation provider is not available.
- JPA delegates validation to the Bean Validation implementation during the `prepersist`, `preupdate` entity life-cycle events.
- Preremove events do not trigger validation, unless enabled via validation group property.

```
<persistence-unit>
    <validation-mode>CALLBACK</validation-mode>
</persistence-unit>
<properties>
    <property name="javax.persistence.validation.group.pre-persist"
        value="javax.validation.groups.Default"/>
    <property name="javax.persistence.validation.group.pre-update"
        value="javax.validation.groups.Default"/>
    <property name="javax.persistence.validation.group.pre-remove"
        value="javax.validation.groups.Default"/>
</properties>
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

One of the goals of Bean Validation is to move validation from the presentation layer to the domain model, to avoid duplication of validation code in each layer of an enterprise application. This is sometimes referred to as the DRY principle: "Don't repeat yourself." Rather than clutter the domain model with validation logic, Bean Validation defines a default set of annotations that can be applied as predefined constraints, and also defines an API and metadata model to allow developers to override and extend the annotations.

Bean Validation is currently at version 1.1. The specification provides validation for any EE technology or specification. However, currently only the JPA, JSF, CDI, and JAX-RS specifications have implemented bean validation into their life cycles.

Before version 1.1, Bean Validation could be applied only to fields or properties (getter methods). Bean Validation 1.1 provides the capability of adding constraints to the parameters that are passed in a method and the values that a method returns. This capability applies also for constructors.

Bean Validation for JSF is enabled by default and can be turned off through a context parameter in the `web.xml` file:

```
<context-param>
    <param-name>
        javax.faces.validator.DISABLE_DEFAULT_BEAN_VALIDATOR
    </param-name>
    <param-value>true</param-value>
</context-param>
```

Bean Validation for JSF beans is invoked automatically for every user-specified validation constraint whenever the components are normally validated.

Bean Validation can be invoked using an instance of a Validator on any Java class, class method, or constructor.

In addition to primitive and wrapper types, Bean Validation can be applied to collections, arrays, and Iterable fields and properties.

Using Bean Validation Constraints

Bean Validation Constraints are applied to Entity Attributes

- Custom Error Messages can be applied:
 - Directly, using message attribute
 - Via the ValidationMessages.properties file
- Default or Custom messages can be used.
- Expressions can reference values.
- Valid annotation triggers validation

```
javax.validation.constraints.Size.message=
Size must be between {min} and {max}
product.bestBefore=
Product Best Before Date must be in the future

@NotNull(message="Discount must be applied")
private BigDecimal discount;
@Size(min=2,max=100)
private String description;
@Future(message="{product.bestBefore}")
private Date bestBefore;

public void updateProduct(@Valid Product p) {...}
```



Examples of Bean Validation Constraints:

```
@AssetFalse
private boolean isSomething;
@AssertTrue
private boolean isSomethingElse;
@DecimalMax(value="999.99")
@DecimalMin(value="0.99")
private BigDecimal price;
@Digits(integer=4, fraction=2)
private BigDecimal discount;
@Size(min=2, max=100)
private String description;
@Max(10)
@Min(1)
private Integer quantity;
@NotNull
@Null
@Future
private Date bestBefore;
@Past
private Date dateManufactured;
@Pattern(regexp="\\" (\d{3})\\)\d{3}-\\d{4}")
private String phoneNumber;
```

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Some Bean Validation annotations do not have required attribute elements (for example, @NotNull, @Null, @Past and @Future).

Note: Each of the annotations also has an @<Annotation>.List type that can be used to define a set of constraints, as in the following example:

```
@Pattern.List( {
    @Pattern-regexp="[A-Z0-9_ %+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}",
    @Pattern-regexp=".?emmanuel.?"
})
```

Each element in the array is processed by the Bean Validation implementation as regular constraint annotations. This means that each constraint in the array is applied to the target. The targets of the constraint list should be of the same type as the initial constraint.

ConstraintValidationException will contain a message from the violated constraints. The default messages are declared in the implementation.

The advantage of the ValidationMessages.properties file is the ability to localize the messages.

See the JSR 349 specification for a complete list of standard error messages.

Constraint violation messages can use Expression Language (enclosed between \${}), as shown in the third code snippet in the slide.

The properties listed below are resolved by the default message interpolator:

```
javax.validation.constraints.AssertFalse.message=must be false
javax.validation.constraints.AssertTrue.message=must be true
javax.validation.constraints.DecimalMax.message=must be less than
${inclusive == true ? 'or equal to ' : ''}{value}
javax.validation.constraints.DecimalMin.message=must be greater than
${inclusive == true ? 'or equal to ' : ''}{value}
javax.validation.constraints.Digits.message=numERIC value out of bounds
(<{integer} digits>.<{fraction} digits> expected)
javax.validation.constraints.Future.message=must be a future date
javax.validation.constraints.Max.message=must be less than or equal to
{value}
javax.validation.constraints.Min.message=must be greater than or equal to
{value}
javax.validation.constraints.NotNull.message=must not be null
javax.validation.constraints.Null.message=must be null
javax.validation.constraints.Past.message=must be a past date
javax.validation.constraints.Pattern.message=must match the following
regular expression: {regexp}
javax.validation.constraints.Size.message=size must be between {min} and
{max}
```

`@javax.validation.Valid` annotation can be used to trigger entity object validation.

You can also programmatically validate entities using following classes:

```
javax.validation.ConstraintViolation
javax.validation.Validation
javax.validation.Validator
javax.validation.ValidatorFactory
```

Code snippet demonstrating programmatic validation:

```
ValidatorFactory validatorFactory =
Validation.buildDefaultValidatorFactory();
Validator validator = validatorFactory.getValidator();
Set<ConstraintViolation<Product>> violations =
validator.validate(product);
for (ConstraintViolation violation: violations) {
    String message = violation.getMessage();
    ...
}
```

Container Managed Persistence

Entity Manager:

- Handles entity instances by executing select, insert, update, and delete operations
- Is initialized with Persistence Unit configuration
 - Persistence Unit is a set of entities representing data from the same data store
 - Persistence Units are configured via the `persistence.xml` file.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="...>
  <persistence-unit name="ProductPU" transaction-type="JTA">
    <jta-data-source>jdbc/productDB</jta-data-source>
    <properties>
    </persistence-unit>
</persistence>
```

Container Managed (Java EE) deployment:

- Entity Manager is directly injected.
- Container can control transactions.

```
package demos.model;
import javax.persistence.*;
public class ProductManager {
  @PersistenceContext (unitName="ProductPU")
  private EntityManager em;
  ...
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The EntityManager interface performs the work of persisting entities.

- Entities that an entity manager instance holds references to are managed by the entity manager.
- A set of entities within an entity manager at any given time is called its persistence context.
- Entity instances in the persistence context are unique (for example, only one Employee instance with an ID of 110).
- A persistence provider creates the implementation details to read from and write to in a given database.
- An instance of an EntityManager is injected through the `@PersistenceContext` annotation using an optional persistence unit name in the injection.

Persistence Unit defines a set of entity classes that are managed by EntityManager instances in an application. This set of entity classes represents the data contained within a single data store.

Persistence Unit Elements:

- Provider: This is an ORM implementation used behind JPA. Typical choices are EclipseLink or Hibernate. Provider must be an implementation of the `javax.persistence.spi.PersistenceProvider` class.

- class, jar-file or mapping-file: These are elements that describe a set of managed persistence classes, which are managed by a persistence unit. These could be defined by using one or more of the following:
 - Annotated managed persistence classes that are contained in the root of the persistence unit unless the exclude-unlisted-classes element is specified
 - One or more object-relational mapping XML files (orm.xml)
 - One or more JAR files that are searched for the classes
 - An explicit list of classes
- exclude-unlisted-classes: This element excludes all entities that are not explicitly listed in the class, jar-file, or mapping-file property.
- shared-cache-mode-: This element controls whether second-level caching is in effect for the persistence unit.
- validation-mode: This element controls whether automatic life-cycle event time validation is in effect.

Persistence Unit can be configured to use two transaction-type attribute values:

- JTA: EntityManager will support Java Transactional API (JTA).
- RESOURCE_LOCAL: EntityManager will require you to manage transactions through the code.

Note: In a Java EE environment, if this element is not specified, the default is JTA. In a Java SE environment, if this element is not specified, the default is RESOURCE_LOCAL.

Persistence Unit configuration may define additional properties and hints:

- javax.persistence.lock.timeout: Value in milliseconds for pessimistic lock timeout. This is only a hint.
- javax.persistence.query.timeout: Value in milliseconds for query timeout. This is only a hint.
- javax.persistence.validation.group.pre-persist: Groups that are targeted for validation upon the prepersist event (overrides the default behavior)
- javax.persistence.validation.group.pre-update: Groups that are targeted for validation upon the preupdate event (overrides the default behavior)
- javax.persistence.validation.group.pre-remove: Groups that are targeted for validation upon the preremove event (overrides the default behavior)

Example of the persistent.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
  xmlns="http://xmlns.jcp.org/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence"
```

```
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
<persistence-unit name="ProductPU" transaction-type="JTA">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>demos.db.Product</class>
    <jta-data-source>jdbc/productDB</jta-data-source>
    <properties/>
</persistence-unit>
</persistence>
```

Container Managed Transactions with JTA are covered later in the course.

Locally Managed Persistence

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="...>
  <persistence-unit name="ProductPU" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <properties>
      <property name="javax.persistence.jdbc.url" value="..."/>
      <property name="javax.persistence.jdbc.user" value="..."/>
      <property name="javax.persistence.jdbc.driver" value="..."/>
      <property name="javax.persistence.jdbc.password" value="..."/>
    </properties>
  </persistence-unit>
</persistence>
```

Locally Managed (Java SE) deployment:

- Entity Manager is acquired from EntityManagerFactory.
 - Transactions are controlled programmatically.
- ❖ Transaction management is covered in the lesson titled "Implementing Business Logic by using EJBs".

```
EntityManagerFactory emf =
Persistence.createEntityManagerFactory("ProductPU");
EntityManager em = emf.createEntityManager();
try {
  em.getTransaction().begin();
  em.persist(...);
  em.merge(...);
  em.remove(...);
  em.getTransaction().commit();
} catch (Exception e) {
  em.getTransaction().rollback();
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

In a Container Managed Persistence scenario, Persistence provider could be assumed as system default. However, in a Locally Managed scenario Persistence provider needs to be specified.

Also container may provide access to preconfigured datasource; otherwise, database connection properties must be specified.

Example of the persistence.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
  xmlns="http://xmlns.jcp.org/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
  http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
  <persistence-unit name="ProductJPAPU" transaction-
  type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <properties>
      <property name="javax.persistence.jdbc.url"
```

```
value="jdbc:derby://localhost:1527/ProductDB"/>
    <property name="javax.persistence.jdbc.user" value="oracle"/>
    <property name="javax.persistence.jdbc.driver"
value="org.apache.derby.jdbc.ClientDriver"/>
        <property name="javax.persistence.jdbc.password" value="welcome1"/>
        <property name="javax.persistence.schema-
generation.database.action" value="create"/>
    </properties>
</persistence-unit>
</persistence>
```

The UserTransaction interface defines the methods that allow an application to explicitly manage transaction boundaries.

It defines operations to:

`begin()` : Create a new transaction and associate it with the current thread.

`commit()` : Complete the transaction associated with the current thread.

`getStatus()` : Obtain the status of the transaction associated with the current thread.

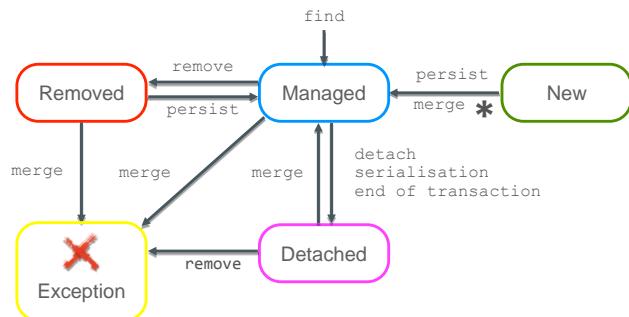
`rollback()` : Roll back the transaction associated with the current thread.

`setRollbackOnly()` : Modify the transaction associated with the current thread such that the only possible outcome of the transaction is to roll back the transaction.

`setTransactionTimeout(int seconds)` : Modify the timeout value that is associated with transactions started by the current thread with the begin method.

Entity Manager Operations

Entity Instance Life Cycle



- flush: Synchronize all managed entities with the database
- refresh: Requery managed entity instance
- contains: Check if entity instance is managed
- clear: Detach all entities

```

package demos.model;
import javax.persistence.*;
public class ProductManager {
    @PersistenceContext (unitName="ProductPU")
    private EntityManager em;
    public void doThings() {
        Product p1 = new Product(7,"Cookie",1.99);
        em.persist(p1);
        Product p2 = em.find(Product.class, 2);
        p2.setPrice(2.99);
        em.remove(p2);
        em.detach(p1);
        p1.setPrice(2.99);
        Product p3 = em.merge(p1);

        em.flush();
        em.refresh(p1);
        boolean isManaged = em.contains(p1);
        em.clear();
    }
}
  
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The life cycle of an entity is depicted in the slide. An entity can have one of several states:

- New: An entity object instance that has no persistent identity and is not yet associated with a persistence context (for example, when an Employee object is created by using the new keyword)
- Managed: An instance with a persistent identity that is currently associated with a persistence context
- Detached: An instance with a persistent identity that is not (or no longer) associated with a persistence context
 - Detached entities are still object instances.
 - They can be read and modified, but they are not persisted to the database.
 - A detached entity is an "offline" entity.
- Removed: An instance with a persistent identity that is associated with a persistence context and that will be removed from the database on transaction commit

Entities can become detached in a number of ways:

- When the transaction associated with a transaction-scoped persistence context commits
- If a stateful session bean with an extended context is removed, all of its managed entities become detached.
- If the `clear()` method is used on an entity manager, all entities associated with that entity manager become detached.
- If the `detach(entity)` method is used, the entity specified becomes detached.

- When a transaction rollback exception occurs, all entities in all of the persistent contexts associated with the transaction become detached.
- When an entity is serialized, the serialized form of the entity is detached.

Managed entities do not need merge to be updated.

Detached entities may accumulate changes, before they are merged.

* When the merge method is used on an entity that does not already exist, a new entity is created as a copy of the entity passed to the method. The entity returned from the method is managed:

```
Product p1 = new Product(7, "Cookie", 1.99);  
Product p2 = em.merge(p1);
```

To locate an entity in the database, the EntityManager will locate a row in the database based on the primary key—the equivalent of a SQL SELECT statement.

```
Product p2 = em.find(Product.class, 2);
```

The find method uses the entity class that is passed in the first argument to determine what type to use for the primary key and for the return type. (Therefore, an extra cast is not required.)

When the call returns, the Product instance is a managed entity.

If no Product with the ID 2 is found, a null is returned.

Locking and @Version

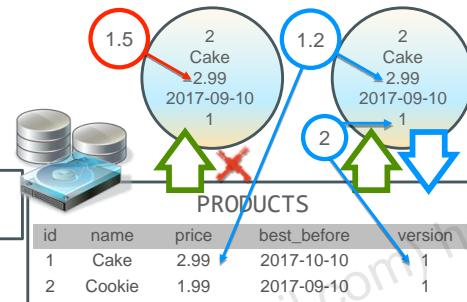
Locking Modes and Version attributes are designed to ensure data integrity and support different levels of concurrency.

```
package demos.db;
import javax.persistence.*;
@Entity
@Table(name="PRODUCTS")
public class Product implements Serializable {
    @Id
    @NotNull
    private Integer id;
    ...
    @Version
    private Integer version;
    ...
}
```

incremented every time record is updated

Locking Modes

| | Pessimistic | Optimistic |
|--------------------------|-------------|------------|
| Read | LOCK | |
| Write | | LOCK |
| find entity | LOCK | |
| update or remove entity | | LOCK |
| post changes to database | | LOCK |



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The example above assumes that two transactions attempt to update products 2 Cake price to 1.5 and 1.2. Only the transaction that posted changes first succeeds. Version column value is incremented and another transaction is prevented from saving changes.

`LockModeType` Enum defines locking mechanism supported by `EntityManager`

Possible values are:

- `NONE`: No lock will be performed.
- `OPTIMISTIC`
 - Optimistic lock places the lock on a record only when record is posted to the database.
 - Version values in the entity instance and in the database record are compared when record is updated.
 - If version value is not the same, it is assumed that record has been changed by another transaction. Current Transaction will rollback and `OptimisticLockException` will be thrown.
 - If version value matched, then version is incremented when the record is updated can be committed.
- `OPTIMISTIC_FORCE_INCREMENT`
 - Similar to the Optimistic mode
 - Version update that occurs even if the entity has not been updated
 - Useful when working with entities that form relationships; for example, changing version of the Order if you modified any of its Items

- PESSIMISTIC_FORCE_INCREMENT: Pessimistic write lock, with version update. This option is used to achieve consistent behavior when same tables are used in both pessimistic and optimistic locking strategies.
- PESSIMISTIC_READ
 - Depending on the database type, JPA provider may switch to PESSIMISTIC_WRITE mode instead.
 - Locks the record with read lock state when you read the record. This prevents other transactions from changing the record, but still allows them to read it
- PESSIMISTIC_WRITE
 - Locks the record, preventing other transaction from both reading and writing it
- READ: **Synonymous with OPTIMISTIC**
- WRITE: **Synonymous with OPTIMISTIC_FORCE_INCREMENT**

Version annotation specifies the version field or property of an entity class that serves as its optimistic lock value. The version is used to ensure integrity when performing the merge operation and for optimistic concurrency control.

Only a single Version property or field should be used per class; applications that use more than one Version property or field will not be portable.

The Version property should be mapped to the primary table for the entity class; applications that map the Version property to a table other than the primary table will not be portable.

The following types are supported for version properties: int, Integer, short, Short, long, Long, java.sql.Timestamp.

Note: Timestamp stratagem is not reliable.

Changing Locking Mode

Optimistic Lock is a default option. Locking mode can be changed:

- For a specific Entity Instance in managed state using operations:
 - find
 - refresh
 - lock
- When defining a JPQL or SQL query programmatically or via annotation

```
@PersistenceContext  
EntityManager em;  
...  
Product p1 = ...;  
  
p1 = em.find(Products.class, 42, LockModeType.PESSIMISTIC_WRITE);  
  
em.refresh(p1, LockModeType.OPTIMISTIC_FORCE_INCREMENT);  
  
em.lock(p1, LockModeType.OPTIMISTIC);  
  
Query q = em.createQuery(...);  
q.setLockMode(LockModeType.PESSIMISTIC_FORCE_INCREMENT);  
  
@NamedQuery(name="lockProductQuery",  
query="SELECT p FROM Product p  
WHERE p.name LIKE :name",  
lockMode=PESSIMISTIC_READ)
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Java Persistence Query Language (JPQL)

Define JPQL statements

- Refer to Java Object names.
- Use bind parameters.
 - Named :name
 - Positional ?1
- Improve performance by using bulk actions instead of selecting, updating, or deleting individual entities one by one.
- They can be annotated or created programmatically.

```

@Entity
@Table(name="PRODUCTS")
@NamedQueries({
    @NamedQuery(name="Product.findByDateAndPrice",
        query="SELECT p.name, p.price FROM Product p
            WHERE p.bestBefore > :someDate
            AND p.price BETWEEN :minPrice AND :maxPrice"),
    @NamedQuery(name="Product.updateOld",
        query="UPDATE Product AS c SET p.price=p.price*0.5
            WHERE p.bestBefore < ?1"),
    @NamedQuery(name="Product.removeOld",
        query="DELETE FROM Product c
            WHERE p.bestBefore > :someDate") })
public class Product implements Serializable {
    @Id
    @NotNull
    private Integer id;
    private String name;
    private BigDecimal price;
    @Temporal(TemporalType.DATE)
    @Column(name="best_before")
    private LocalDate bestBefore;
    ...
}
  
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The Java Persistence Query Language is a query specification language for dynamic queries and for static queries expressed through metadata. JPQL can be compiled to a target language (such as SQL) of a database or other persistent store. This feature enables the execution of queries to be shifted to the native language facilities provided by the database, instead of requiring queries to be executed on the runtime representation of the entity state. As a result, queries can be optimized as well as portable.

JPA supports two methods for constructing queries:

- Query languages, such as the Java Persistence Query Language (JPQL)
- Native SQL

JPQL is a database-independent query language that operates over the logical entity model rather than the physical data model.

JPQL and Native SQL Statements can be constructed via annotations or programmatically with the help of Criteria API.

All JPQL statements reference Java object and attribute names, rather than database names.

SELECT can also be used to return scalar data using a scalar expression or function, as in the example, `SELECT p.price*0.9 FROM Product p`.

UPDATE and **DELETE** statements can greatly improve performance, when bulk actions are required.

Note that the type of operand depends on the expression operation:

- `=, >, >=, <, <=, <>`: Arithmetic
- [NOT] BETWEEN: Arithmetic, string, date, time
- [NOT] LIKE: String
- [NOT] IN: All types
- IS [NOT] EMPTY: Collection
- [NOT] MEMBER OF: Collection
- [NOT] LIKE: String

Note: The LIKE operator operates on a string expression. The string expression must have a string value. The pattern value is a string literal or a string-valued input parameter in which an underscore (`_`) stands for any single character, a percent (%) character stands for any sequence of characters including an empty sequence, and all other characters stand for themselves.

Using JPQL with non-Entity classes

JPQL statements can query data into non-Entity objects

- Use NEW operator to create instances of object that is not an Entity
- Populate each of these objects with data returned by the query
- Often can be used to represent information from different Entities that needs to be joined

```
@NamedQuery(name="FindProductOffers",
    query="SELECT NEW demos.ProductOffer(p.id, p.name, p.price, o.discount)
        FROM Product p JOIN p.offer o")
```

```
package demos;
public class ProductOffer {
    private int id;
    private String name;
    private float price;
    private float discount;
    public ProductOffer(int id, String name, float price, float discount) {...}
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Note that NEW operator requires a relevant constructor name to be fully qualified.

If you use NEW operator to select an Entity then the resulting Entity object would be in a "new" state. This coding technique may be a bit controversial, since a simple object instantiation could have been used to achieve same result.

Executing JPQL Statements

Executing named queries

- If query returns multiple records, use:
 `.getResultList();`
- If query returns a single record, use:
 `.getSingleResult();`
- To update or delete records, use:
 `.executeUpdate();`

```
@PersistenceContext
public EntityManager em;
...
List<Product> products =
em.createNamedQuery("Product.findByDateAndPrice")
.setParameter("someDate", LocalDate.now())
.setParameter("minPrice", 1)
.setParameter("maxPrice", 10)
.getResultList();
```

JPQL statements can be defined dynamically.

- Use the `createQuery` method to define `TypedQuery` Object.

```
TypedQuery<Product> query =
em.createQuery("SELECT p.name FROM Product p
WHERE p.price > :value",
Product.class);
query.setParameter("value", 5);
List<Product> products = query.getResultList();
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Use StoredProcedureQuery object to execute database stored procedures.

You may use Native SQL statements instead of JPQL with

`@NativeQuery` annotation or `em.createNativeQuery` method.

However, because native queries use direct database-specific SQL, they are less portable than JPQL statements.

Query can be conducted dynamically using JPQL statements, or Query Criteria API. Here is an example of a query that retrieves product with price greater than 5 and name starting with "C":

```
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Product> cq = builder.createQuery(Product.class);
Root<Product> product = cq.from(Product.class);
cq.where(builder.gt(product.get(Product_.price), 5).and(builder.like(product.get(Product_.name), "C%")));
Query q = em.createQuery(cq);
q.getResultList();
```

For more information please reference <https://docs.oracle.com/javaee/7/tutorial/persistence-criteria003.htm>

Query object can be used to implement pagination - i.e. getting results in sets of records. Here is an example code that retrieves first 10 records:

```
Query q = em.createNamedQuery("Product.findProducts");  
q.setFirstResult(1);  
q.setMaxResult(10);  
q.getResultList();
```

Summary

In this lesson, you should have learned how to manage persistence by using the Java Persistence API, including how to:

- Create JPA entities with object-relational mappings
- Use Entity Manager to perform database operations with JPA entities
- Handle entity data with conversions, validations, and key generation
- Describe persistence management and locking mechanisms
- Create and execute JPQL statements



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Practice Overview

This practice covers the following tasks:

- Create JPA Entity Product mapped to the Database Table Products
- Defining attribute handlings
 - Generate primary key from sequence
 - Create datatype converter
 - Add validations constraints
- Adding JPQL queries
- Creating Java SE JPA client
 - Using EntityManager to handle Entity operations
 - Control Transactions



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Implementing Business Logic by Using EJBs

4



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



ORACLE®

Objectives

This lesson teaches how to implement business logic with Enterprise JavaBeans. After completing this lesson, you should be able to:

- Create Session EJB components
- Create EJB business methods
- Manage EJB life cycle with container callbacks
- Use asynchronous EJB operations
- Control transactions
- Create EJB timers
- Create and apply interceptors



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



EJBs and EJB Container

Enterprise JavaBean

- Server-side component that encapsulates the business logic of an application

EJB container

- An environment in which EJBs are executed
- Provides system-level services, such as transactions and security, to the enterprise beans

Types of EJB containers:

- Full implementation
- Embeddable

```
import javax.ejb.embeddable.*;  
...  
EJBContainer container = EJBContainer.createEJBContainer();  
Context ctx = container.getInitialContext();  
SomeBean foo = (SomeBean)ctx.lookup("java:global/SomeBean");  
...  
container.close();
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Features of EJBs:

1. Reusable: The application assembler can build new applications from existing beans
2. Portable: Use the standard APIs, these applications can run on any compliant Java EE server.
3. Easy to Develop and use: As the EJB container is responsible for system-level services, such as transaction management and security authorization, the bean developer can concentrate on solving business problems.

You should consider using enterprise beans if your application requires scalability, data integrity, and variety of clients.

The EJB container provides the following functionality:

- Encapsulates access to external resources such as databases and legacy systems
- Manages the life cycles of instances of the EJB component's implementation class
- Isolates the class that provides the implementation from its clients
- Provides timer services, and can invoke methods at certain times, which allows the EJB component to perform scheduled tasks
- Monitors, for message-driven beans, a message queue on behalf of the EJB component

Embedded EJB Container

Most of the services present in the enterprise bean container in a Java EE server are available in the embedded enterprise bean container, including injection, container-managed transactions, and security. The enterprise bean components execute similarly in both embedded and Java EE environments and, therefore, the same enterprise bean can be easily reused in both stand-alone and networked applications.

Enterprise JavaBean Types

Enterprise JavaBean classification:

| Message-Driven | | |
|--|---|---|
| Act as a message consumer for a Queue or a Topic | | |
| Session | | |
| Stateful | Stateless | Singleton |
| Perform tasks that require to hold client specific information across method invocations | Perform tasks that do not require to retain any client specific context across method calls | Perform tasks that use information shared across the entire application |
| | May be exposed as a WebService | May be exposed as a WebService |



❖ Message-Driven Beans are covered in the next lesson



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Session EJBs

There are three types of EJB Session Beans:

- Stateless
 - Annotated with `@Stateless`
 - Not associated with any particular client
- Stateful
 - Annotated with `@Stateful`
 - One bean instance per client
- Singleton
 - Annotated with `@Singleton`
 - One bean instance per JVM

Optionally, inject `EJBContext` object that provides:

- Access to security properties
- Context data, such as Interceptor or WebService context
- Transaction control

```
package demos;
import javax.ejb.*;
@Stateless
public class ABean { }
```



```
package demos;
import javax.ejb.*;
@Stateful
public class BBean implements Serializable { }
```



```
package demos;
import javax.ejb.*;
@Singleton
public class CBean { }
```

```
@Inject
private EJBContext context;
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

By separating the presentation from the business logic, EJBs can serve the needs of a presentation object and their life cycle can be managed by another process.

- Scalability through pooling
- Transaction management
- Injection, rather than instantiation
- Services can be swapped out without rewriting presentation code

The Session Bean class must:

- Be top-level class
- Be public class
- Not be final
- Not be abstract
- Have a public constructor that takes no parameters
- Not define the `finalize` method
- Implement the methods of the bean's business interfaces, if any

Stateless Session Beans

- The bean does not retain client-specific information.
- A client might not use the same session bean instance during subsequent method calls.
- A large number of clients can be handled at the same time.

Stateful Session Beans

- Beans belong to a particular client for an entire conversation.
- The client connection exists until the client removes the bean or the session times out.
- The container maintains a separate EJB object and EJB instance for each client.

There is always a cost to maintain client state; maintaining more state than what is needed or using stateful session beans when a stateless bean would be adequate, negatively impacts performance. If an application must store client state, stateful session beans are an effective way to store the client state.

Singleton Session Beans

The EJBContext interface provides an instance with access to the container-provided runtime context of an enterprise bean instance.

This interface is extended by the SessionContext, EntityContext, and MessageDrivenContext interfaces to provide additional methods specific to the enterprise interface bean type.

For more information on EJBContext see:

<https://docs.oracle.com/javaee/7/api/javax/ejb/EJBContext.html>

Accessing Session Beans

An Session bean can have different views (ways of accessing the bean):

- **Remote:** Used by remote invokers (all parameters and returned values are serialized)
- **Local:** Used by local invokers
- **WebService:** Used by external web service invokers
- No Interface: Used by local invokers (All public methods are automatically exposed.)

```
@Stateless
@Remote(ProductFacadeRemote)
@Local(ProductFacadeLocal)
@WebService
@LocalBean
public class ProductFacade
    implements ProductFacadeLocal,
               ProductFacadeRemote {
    public Product findProduct(int id){...}
    public void createProduct(Product product){...}
    @WebMethod
    public void removeProduct(int id){...}
    public void updateProduct(Product product){...}
}
```

```
@Remote
public interface ProductFacadeRemote {
    public Product findProduct(int id);
    ...
}

@Local
public interface ProductFacadeLocal {
    public void createProduct(Product product);
    ...
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Remote Interface

A session bean that is implemented with a remote interface is accessible by components:

- Within the application
- Outside of the application
- Co-resident in the JVM
- On a different JVM (most likely)

Session beans with remote capabilities simplify the development of large-scale distributed solutions. However, there are potential challenges with remote beans, such as responsiveness and bandwidth. Remote beans are typically accessed by clients outside of the container and they act as an entry-point to a complex middleware solution.

Local Interface

A session bean that is implemented with a local interface is accessible only by the components and resources that are packaged within the application that is running on the same JVM.

Implementing session beans by using a local interface has several advantages:

- Local interfaces create encapsulation, in the same manner as access modifiers.
- Hiding certain components from other resources may be beneficial when you are dealing with sensitive algorithms.

Local interfaces adopt a pass-by-reference semantic, which is more efficient than remote communication.

No-Interface Implementation

Starting with EJB 3.1, EJB components can be developed without the creation of a separate business interface. This implementation strategy is known as a no-interface implementation.

- This involves a less cumbersome development process; however, it does have certain side effects.
- To create a no-interface implementation of a session bean, apply the @LocalBean annotation directly to the bean class, or assume the default local client access mode.
- When using the no-interface representation of a session bean, all the public methods defined within the bean are considered part of the local interface.

Web Service Implementation

The Stateless session and Singleton session beans may have web service clients. A web service client accesses a session bean through the web service client view. The web service client view is described by the WSDL document for the web service that the bean implements.

The web service client view of an enterprise bean is location-independent and remotable. Web service clients may be Java clients or clients not written in the Java programming language. A web service client that is a Java client accesses the web service by means of the JAX-WS or JAX-RPC client APIs. Such EJB classes must be annotated with either the javax.jws.WebService or the javax.jws.WebServiceProvider annotation and business methods that are exposed to web service clients must be annotated with javax.jws.WebMethod.

Note: Remote and Local interfaces can contain same operations. However, you need to remember that unlike local interface, remote interface parameters and return values are always serialized. This means that Local and Remote interface operation semantics may be quite different.

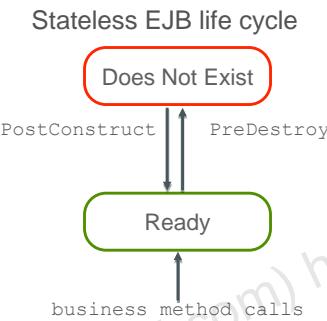
Note: It is advisable to package Remote interface and related classes (those that are used as parameters and return types) into a separate archive, so it could be distributed on to both client and server tiers.

Stateless Session Bean Life Cycle

EJB Container instantiates stateless session EJBs as pooled instances, or when the caller injects or looks up the bean.

- The life cycle callback methods are annotated with : @PostConstruct and @PreDestroy annotations
- A @PostConstruct annotated method is invoked when a bean instance is created
- A @PreDestroy annotated method is invoked when the bean instance is destroyed and the bean's instance is then ready for garbage collection.

```
package demos;
import javax.ejb.*;
@Stateless
public class SomeBean {
    @PostConstruct
    public void init() {...}
    @PreDestroy
    public void cleanup() {...}
    public void doThings() {...}
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The stateless session bean life-cycle diagram shown in the slide provides the context for the stateless session bean life-cycle events. Stateless session beans have a life cycle that mirrors the common life cycle that was previously described.

A stateless session bean generates the following lifecycle events:

- PostConstruct
- PreDestroy

Instantiation of a Stateless Session Bean:

Stateless session beans experience instantiation on the first client request for the session bean in the naming system. That is, when a client uses dependency injection or the more traditional JNDI mechanism to look up a session bean, the container determines whether a stateless session bean exists, and if one does not, it creates one.

Most EJB containers use a pooling strategy for session beans. Pooling of session beans allows the container to reuse objects that are "idle" instead of creating new objects, handling the request, and then destroying them.

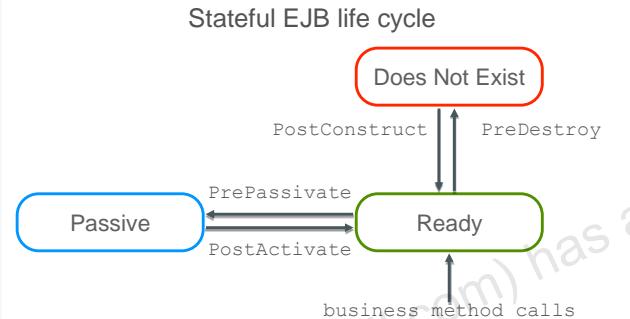
Also remember that stateless session beans are not tied to a specific client. As a result, subsequent "lookups" of a stateless session bean in the EJB container do not automatically translate into subsequent (and additional) new instance creation.

Stateful Session Bean Life Cycle

Container allocates a Stateful session bean instance per caller.

- PostConstruct and PreDestroy work in the same way as in the Stateless session bean.
- PrePassivate is used when bean instance is swapped out of memory.
- PostActivate is used when bean instance is restored to memory.
- Passivation/Activation can be turned off with the passivationCapable=false attribute.
- Remove is used to define a business method that client call to request this bean instance to be removed.

```
package demos;
import javax.ejb.*;
@Stateful(passivationCapable=true)
public class SomeBean implements Serializable {
    @PostConstruct
    public void init() {...}
    @PreDestroy
    public void cleanup() {...}
    @PostActivate
    public void restore() {...}
    @PrePassivate
    public void save() {...}
    public void doThings() {...}
    @Remove
    public void remove() {...}
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Stateful Session Bean Operational Characteristics

With stateful session beans:

- The bean belongs to a particular client for an entire conversation
- The client connection exists until the client removes the bean or the session times out
- The container maintains a separate EJB object and EJB instance for each client

Comparison of Stateless and Stateful Behavior

Session beans can be stateless or stateful. The 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.

Alternatively, stateful services require that information that is obtained during one method invocation be available during subsequent method calls:

- Shopping carts
- Multipage data entry
- Online banking

Note: Stateful session beans are by default passivation capable and, therefore, there is no need to use the passivationCapable attribute, unless you want to switch passivation off. Alternatively, you can use ejb-jar.xml deployment descriptor to control this property.

Note: If Enterprise JavaBeans are invoked from the web tier, it is likely that state will be maintained by the HttpSession object in a web container. Therefore, EJB components can actually be stateless.

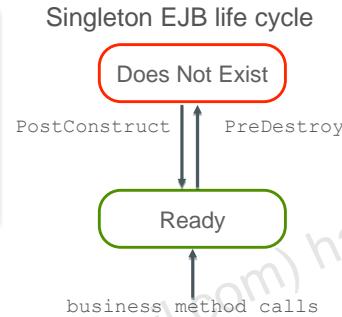
Singleton Session Bean Life Cycle

Container allocates one Singleton session bean instance shared between all callers.

- PostConstruct and PreDestroy work in the same way.
- Lock annotation is used to prevent concurrent callers from corrupting Singleton bean instance state.
 - javax.ejb.LockType.WRITE (default) gives caller exclusive access to the singleton.
 - javax.ejb.LockType.READ allows concurrent access —should be used for read-only operations.
 - javax.ejb.AccessTimeout controls maximum amount of time caller waits for the lock to be released
- The Startup annotation instructs the container to perform eager initialization of the EJB.

```
package demos;
import javax.ejb.*;
@Singleton
public class SomeBean {
    @PostConstruct
    public void init() {...}
    @PreDestroy
    public void cleanup() {...}
    @Lock(LockType.READ)
    public void doThings() {...}
    @AccessTimeout(value = 5, unit = TimeUnit.SECONDS)
    public void doOtherThings() {...}
}
```

```
package demos;
import javax.ejb.*;
@Startup
@Singleton
public class SomeBean {
    ...
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Singleton session beans have a life cycle that mirrors the common session bean lifecycle transitions. In fact, the life cycle of a singleton bean is very similar to that of a stateless bean.

A singleton session bean generates the following life-cycle transitions:

- Instantiation: A singleton session bean transitions from the "Does not exist" state to the "Ready" state as part of the instantiation process.
- Ready: Unlike other session beans, clients can concurrently access the business methods of a singleton bean.
- PreDestroy Callback: There is no specific way for a client to initiate removal of a singleton session bean. Removal is determined by the container as part of the application shutdown process.

A singleton session bean has the following characteristics:

- The container maintains a single EJB instance per JVM. Typically, applications utilize a single JVM; however, in a clustered environment; there can be multiple singleton instances.
- The bean can belong to multiple clients simultaneously.
- Concurrent access is controlled by using Container Managed Concurrency by default.

Unlike stateless session beans, a singleton session bean maintains state across method calls. Additionally, state is shared by all clients because all clients access the same bean instance.

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 by 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 of the following:

- `@javax.ejb.Lock(javax.ejb.LockType.WRITE)`
- `@javax.ejb.Lock(javax.ejb.LockType.READ)`

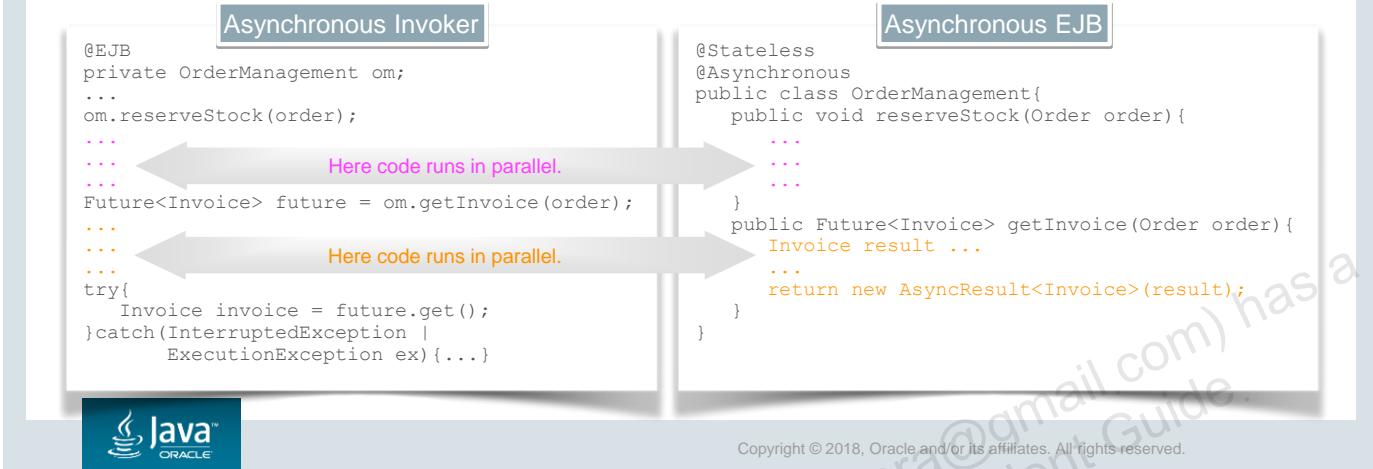
`@javax.ejb.AccessTimeout` annotation can set timeout to:

- A specific period of time
- Or -1 in this case caller would be allowed to wait as long as it takes for the bean instance to become available after lock is released
- Or 0 in this case caller is instructed not to wait for the lock to be released, so the caller would receive an error if the bean instance is currently locked by another caller
- Lock and AccessTimeout annotations can also be used with Stateful Session beans to control access to the bean from the client that can make simultaneous calls to the bean.

Asynchronous EJB operations

Entire EJB or specific operations can be marked with the asynchronous annotation.

- While asynchronous operation is executing its logic, the caller can perform other actions.
- The Future object presents results returned by the asynchronous method.



In the case of an asynchronous method with a void return type, nothing is returned to the client.

However, when an asynchronous method designates a `Future<V>` return type, the method must create an instance of the `Future` interface, and return the instance to the client.

Future Object is a placeholder for the result to be produced by the asynchronous method.

As part of EJB 3.1, a utility class known as `AsyncResult` can be used to represent a Future.

The `javax.ejb.AsyncResult<V>` class is a concrete implementation of the `Future<V>` interface that is provided as a helper class for returning asynchronous results.

Future object defines following operations:

- `cancel(boolean mayInterruptIfRunning)` : Attempts to cancel execution of this task
- `get()` : Waits if necessary for the computation to complete, and then retrieves its result
- `get(long timeout, TimeUnit unit)` : Waits if necessary for at most the given time for the computation to complete, and then retrieves its result, if available
- `isCancelled()` : Returns true if this task was cancelled before it completed normally
- `isDone()` : Returns true if this task is completed

Java Transaction API

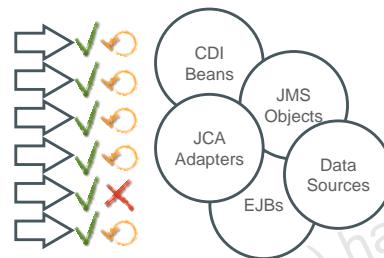
Transactions are **ACID**:

- **Atomic**: Transaction must be done, or undone completely.
- **Consistent**: Transaction brings system from one consistent state to another.
- **Isolated**: Transaction state is not visible to components outside of this transaction.
- **Durable**: Once transaction completes, all changes become permanent.

JTA Transaction Implementations:

- Programmatic - **Bean Managed Transactions (BMT)**
 - Available for all Java Components Types
 - Not recommended for EJBs and CDI Beans
- Declarative - **Container Managed Transactions (CMT)**
 - Available and recommended for EJBs and CDI Beans

Various objects can participate in transactions.



All actions within the transaction have to succeed or must be undone.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The Java EE application developer's concern is with scoping transactions. Scoping can be programmatic (bean-managed transactions) or declarative (container-managed transactions).

Scoping refers to the selection of operations that 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 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 allow only 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.

Note: Java supports only flat transaction model; chained or nested transaction are not supported.

Flat Transaction Model characteristics:

- In a particular thread, only one transaction is in effect at a time.
- This is the only transaction model supported by the Java EE specification and supported by all database vendors.
- Flat transactions also allow for a simple declarative transaction model. More complex transaction models can be simulated in code if necessary.

Programmatic Transactions (BMT)

Enabling Bean Managed Transactions:

- Initialize UserTransaction Object within Java EE Container using @Resource injection.
- For the EJB, set the BEAN TransactionManagement property.
- Initialize UserTransaction Object outside of the Java EE Container using JNDI lookup.

```
@RequestScoped
public class SomeBean {
    @Resource
    private UserTransaction t;
    ...
}

@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class SomeEJB {
    @Resource
    private UserTransaction t;
    ...
}

Context ctx = new InitialContext();
UserTransaction t = (UserTransaction)ctx.lookup("java:comp/UserTransaction");
```

Control transactions using operations:

- begin(): Start transaction
- commit() and rollback(): End transaction

```
try{
    t.begin();
    ...
    t.commit();
} catch(Exception e){
    t.rollback();
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

UserTransaction Object is used to control transactions in Java. It has the following operations:

- begin(): Create a new transaction and associate it with the current thread.
- commit(): Complete the transaction associated with the current thread.
- getStatus(): Obtain the status of the transaction associated with the current thread.
- rollback(): Roll back the transaction associated with the current thread.
- setRollbackOnly(): Modify the transaction associated with the current thread such that the only possible outcome of the transaction is to roll back the transaction, even if commit is invoked.
- setTransactionTimeout(int seconds): Modify the timeout value that is associated with the transactions started by the current thread with the begin() method.

Declarative Transactions (CMT)

Enabling Container Managed Transactions:

- EJB TransactionManagement property is CONTAINER by default .
- CDI Beans use Transactional Annotation.
- Demarcate Transactions boundaries using Transactional Attribute Annotations (default is REQUIRED).

To tell container to rollback:

- EJBs call the setRollbackOnly operation
- CDI Beans throw exceptions (see notes)

```
@RequestScoped
@Transactional
public class SomeBean {
    @Transactional(value = Transactional.TxType.REQUIRED,
        rollbackOn = {list of exceptions},
        dontRollbackOn = {list of exceptions})
    public void someMethod() {
        ...
        // just throw exception to rollback
    }
}
```



```
@Stateless
@TransactionManagement(CONTAINER)
public class SomeEJB {
    @Resource
    private SessionContext ctx;
    @TransactionAttribute
    (TransactionAttributeType.REQUIRED)
    public void someMethod() {
        try{
            ...
        }catch(Exception e){
            ctx.setRollbackOnly();
        }
    }
}
```

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

By default, EJBs use CONTAINER Transactional Management.

EJB Declarative Transactions use the TransactionalAttribute annotation.

The default value for the Stateless EJB is REQUIRED and for Stateful it is REQUIRES_NEW.

CDI Beans Declarative Transactions use javax.transaction.Transactional annotation. It provides application the ability to declaratively control transaction boundaries on CDI-managed beans, as well as classes defined as managed beans by the Java EE specification, at both the class and method level where method level annotations override those at the class level.

TxType.REQUIRED is the default.

By default, checked exceptions do not result in the transactional interceptor marking the transaction for rollback and instances of RuntimeException and its subclasses do. This default behavior can be modified by specifying exceptions that result in the interceptor marking the transaction for rollback and/or exceptions that do not result in rollback.

The rollbackOn element can be set to indicate exceptions that must cause the interceptor to mark the transaction for rollback.

Conversely, the dontRollbackOn element can be set to indicate exceptions that must not cause the interceptor to mark the transaction for rollback.

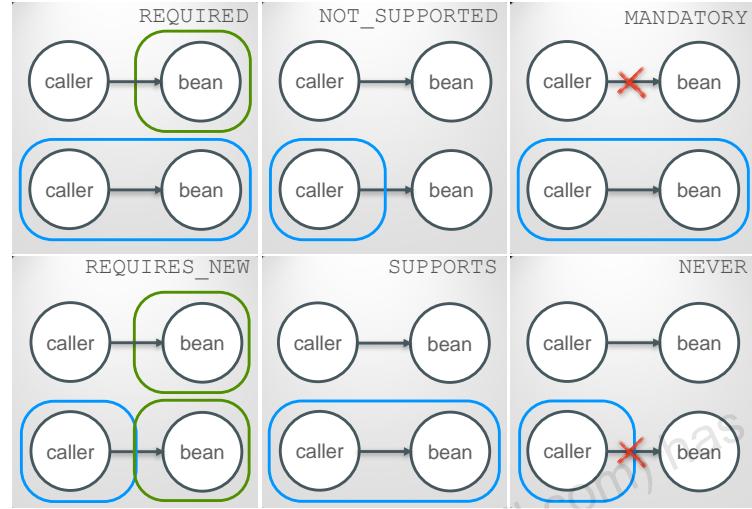
When a class is specified for either of these elements, the designated behavior applies to subclasses of that class as well. If both elements are specified, dontRollbackOn takes precedence.

Java EE 7 also introduced a new CDI scope, @TransactionScoped. This scope allows this object to participate in a transaction propagated across stateless session calls.

Demarcate Transactional Attributes

Depending on the Transactional Attribute value, bean methods can:

- Join transaction of invoker
- Start new transaction
- Run with no transactional context
- Throw exception



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

- 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.
- REQUIRES_NEW:** The method always runs in its own transaction. Any existing transaction is suspended.
- NOT_SUPPORTED:** The method never runs in a transaction. Any existing transaction is suspended.
- 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.
- MANDATORY:** It is an error to call this method outside of a transaction.
- NEVER:** It is an error to call this method in a transaction.

Transaction Scoped Beans

@TransactionScoped Annotation allows CDI Beans to participate in a transaction propagated across stateless session calls.

```
@Stateless
public class OrderManagement {
    @Inject
    private Order order; CDI bean is injected.
    @EJB
    private InvoiceManagement im;
    @TransactionalAttribute(REQUIRES_NEW)
    public void completeOrder() {
        order.calculateTotal();
        im.issueInvoice();
    }
}
```

```
@TransactionScoped
public class Order implements Serializable {
    ...
}
```

The container starts a new transaction.

Use CDI bean in the transaction.

```
@Stateless
public class InvoiceManagement {
    @Inject
    private Order order;
    public void issueInvoice() {
        order.readyForPayment();
    }
}
```

CDI bean is injected.

Continue to use same CDI bean instance in the same transaction.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Java EE 7 also introduced a new CDI scope, @TransactionScoped. This scope allows this object to participate in a transaction propagated across stateless session calls.

Timers

Purpose of timers is to execute code at some point in time or periodically.

- Timers can be applied to Stateless, Singleton, Message-Driven, and 2.1 Entity Beans.
- By default timers are stateful, so they are redelivered after server restart.
- Automatic timers can be configured with annotations or deployment descriptors.

Timer types:

- Programmatic
- Created using TimerService object
- When such timer expires (goes off), the container calls single method annotated @Timeout in the bean's implementation class.
- Automatic
- Created with multiple @Schedule or @Schedules

Timeout methods:

- Must not return values (be declared with void return type)
- Optionally take Timer object as the only parameter



```
@Resource
private TimerService timerService;
public void someMethod() {
    timerService.createTimer(...);
}
@Timeout
public void doThings(Timer timer) {...}
```

```
@Schedule(dayOfWeek="Sun", hour="0")
public void doThings() {...}

@Schedule(minute="*/15", hour="9-17")
public void doMoreThings() {...}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The EJB timer service is a container-managed service that provides a way to allow methods to be invoked at specific times or time intervals.

The EJB timer may be set to invoke certain EJB methods at a specific time, after a specific duration, or at specific recurring intervals.

Timers can be created programmatically by interacting with TimerService or automatically through annotations.

These Timer tasks apply to the following types of enterprise beans:

- Stateless and singleton session beans
- Message-driven beans
- 2.1 entity beans, except calendar-based timer and nonpersistent timer functionality that are not supported for 2.1 Entity beans.

Timers are intended for long-lived business processes and are by default persistent.

Types of timers:

- Programmatic timers are defined:
 - By calling one of the timer creation methods of the TimerService interface
 - When Timer object expires. EJB container triggers an EJB method that is marked with Timeout annotation.
- Automatic timers are created on successful deployment of an enterprise bean that contains methods that are annotated with the java.ejb.Schedule or java.ejb.Schedules annotations, or configured with ejb-jar.xml.

In the event of the application server shutdown, timers behave differently, depending on their persistency type:

- Persistent Timer (default)
 - A timer is guaranteed to survive application server crashes and shutdowns. Timers that expire during application server shutdowns are redelivered on server restart. In most cases, the application server, on power up, notifies the enterprise bean of all expired timers.
- Nonpersistent Timer
 - Nonpersistent timers have a life span that is tied to the life span of the JVM in which it is created. However, they are canceled upon server shutdown, crash, and so on. They can be created again programmatically or automatically.

Calendar-Based Timer Expressions

Timer expressions can be used within:

- @Schedule annotation
- ScheduleExpression Object
- The ejb-jar.xml file

| Attribute | Allowed Values | Default | Examples |
|-------------------|--|---------|---|
| second | 0 to 59 | 0 | second="30" |
| minute | 0 to 59 | 0 | minute="15" |
| hour | 0 to 23 | 0 | hour="13" |
| dayOfWeek | 0 to 7 (both 0 and 7 refer to Sunday) Sun, Mon, Tue, Wed, Thu, Fri, Sat | * | dayOfWeek="3" dayOfWeek="Mon" |
| dayOfMonth | 1 to 31 -7 to -1 (a negative number means the nth day or days before the end of the month) Last [1st, 2nd, 3rd, 4th, 5th, Last] [Sun, Mon, Tue, Wed, Thu, Fri, Sat] | * | dayOfMonth="15" dayOfMonth="-3" dayOfMonth="Last" dayOfMonth="2nd Fri" |
| month | 1 to 12 Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec | * | month="7" month="July" |
| year | A four-digit calendar year | * | year="2010" |



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Setting an attribute to an asterisk symbol (*) - wildcard, represents all allowable values for the attribute.

- The following expression represents every minute: minute="*"
- The following expression represents every day of the week: dayOfWeek="*"

To specify two or more values for an attribute, use a comma (,) to separate the values. A range of values is allowed as part of a list. Wildcards and intervals, however, are not allowed.

Duplicates within a list are ignored.

- The following expression sets the day of the week to Tuesday and Thursday: dayOfWeek="Tue, Thu"
- The following expression represents 4:00 a.m., every hour from 9:00 a.m. to 5:00 p.m. using a range, and 10:00 p.m.: hour="4,9–17,22"

Use a dash character (-) to specify an inclusive range of values for an attribute. Members of a range cannot be wildcards, lists, or intervals. A range of the form x–x, is equivalent to the single-valued expression x. A range of the form x–y where x is greater than y is equivalent to the expression x–maximum value, minimum value–y. That is, the expression begins at x, rolls over to the beginning of the allowable values, and continues up to y.

- The following expression represents 9:00 a.m. to 5:00 p.m.: hour="9–17"
- The following expression represents Friday through Monday: dayOfWeek="5–1"
- The following expression represents the 25th day of the month to the end of the month, and the beginning of the month to the fifth day of the month: dayOfMonth="25–5"
- It is equivalent to the following expression: dayOfMonth="25–Last,1–5"

The forward slash (/) constrains an attribute to a starting point and an interval and is used to specify every N seconds, minutes, or hours within the minute, hour, or day. For an expression of the form x/y, x represents the starting point and y represents the interval. The wildcard character may be used in the x position of an interval and is equivalent to setting x to 0.

- Intervals may be set only for second, minute, and hour attributes.
- The following expression represents every 10 minutes within the hour: minute="*/10"
- It is equivalent to: minute="0,10,20,30,40,50"
- The following expression represents every 2 hours starting at noon: hour="12/2"

Define Programmatic Timers

Programmatic Timers allow developers to create Timer Objects dynamically.

- **TimerService** object controls creation of timers, with various methods, such as:
 - createCalendarTimer
 - createIntervalTimer
 - createSingleActionTimer
 - createTimer
- Time may be triggered as a one-off or periodic event, or both.
- Timers may use Schedule Expressions.
- **Timeout** annotation designated the method that timer should trigger.
- Timer persistency can be switch off:
 - TimerConfig.setPersistent(false)

```
@Stateless
public class SomeBean {
    @Resource
    private TimerService timerService;

    @PostConstruct
    public void init() {
        ScheduleExpression e = new ScheduleExpression();
        e.second("30").minute("*/10").hour("*");
        timerService.createCalendarTimer(e);
    }
    public void createSimpleTimer(Date date){
        timerService.createSingleActionTimer(date,
                                             new TimerConfig());
    }
    @Timeout
    public void someMethod(Timer timer) {...}
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

TimerService can be obtained from EJB SessionContext object:

```
@Stateless
public class HelloWorldBean implements HelloWorldRemote {
    @Resource
    private SessionContext sessionContext;
    public void doThings() {
        TimerService timerService = sessionContext.getTimerService();
        timerService.createTimer(...);
        ...
    }
}
```

or via direct resource injection of the TimerService object:

```
@Stateless
public class HelloWorldBean implements HelloWorldRemote {
    @Resource
    private TimerService timerService;
    public void doThings() {
        timerService.createTimer(...);
        ...
    }
}
```

EJB Timer Service allows stateless session beans, singleton session beans, message-driven beans to be registered for timer callback events at a specified time, after a specified elapsed time, after a specified interval, or according to a calendar-based schedule with the following operations:

`createCalendarTimer(ScheduleExpression schedule)` : Create a calendar-based timer based on the input schedule expression.

`createCalendarTimer(ScheduleExpression schedule, TimerConfig timerConfig)` : Create a calendar-based timer based on the input schedule expression.

`createIntervalTimer(Date initialExpiration, long intervalDuration, TimerConfig timerConfig)` : Create an interval timer whose first expiration occurs at a given point in time and whose subsequent expirations occur after a specified interval.

`createIntervalTimer(long initialDuration, long intervalDuration, TimerConfig timerConfig)` : Create an interval timer whose first expiration occurs after a specified duration, and whose subsequent expirations occur after a specified interval.

`createSingleActionTimer(Date expiration, TimerConfig timerConfig)` : Create a single-action timer that expires at a given point in time.

`createSingleActionTimer(long duration, TimerConfig timerConfig)` : Create a single-action timer that expires after a specified duration.

`createTimer(Date initialExpiration, long intervalDuration, Serializable info)` : Create an interval timer whose first expiration occurs at a given point in time and whose subsequent expirations occur after a specified interval.

`createTimer(Date expiration, Serializable info)` : Create a single-action timer that expires at a given point in time.

`createTimer(long initialDuration, long intervalDuration, Serializable info)` : Create an interval timer whose first expiration occurs after a specified duration, and whose subsequent expirations occur after a specified interval.

`createTimer(long duration, Serializable info)` : Create a single-action timer that expires after a specified duration.

`getAllTimers()` : It returns all active timers associated with the beans in the same module in which the caller bean is packaged.

`getTimers()` : It returns all active timers associated with this bean.

Timer Identification

If you associate multiple timers with an enterprise bean, you must also apply a strategy to identify the timer that issues the notification. You can use the `info` parameter of the `createTimer` method to develop your identification and response strategy.

Define Automatic Timers

Automatic Timers allow developers to create a number of Schedules to trigger different operations upon timer expiration.

- EJB Container creates automatic timers upon successful deployment of an enterprise bean that contains one or more method that is annotated with the `java.ejb.Schedule` or `java.ejb.Schedules` annotations, or describes automatic timers in the `ejb-jar.xml` file.
- An info element can contain any serializable information that you want to associate with this timer.
- Timer persistency can be switched off using the `persistent=false` attribute.

```
@Singleton
public class SomeBean {
    @Schedules({
        @Schedule(hour="*", minute="*", second="*/10", persistent=false),
        @Schedule(hour="*", minute="*", second="*/45")
    })
    public void doThings() {...}

    @Schedule(minute="*/15", hour="9-17", info="Something")
    public void doMoreThings() {...}
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

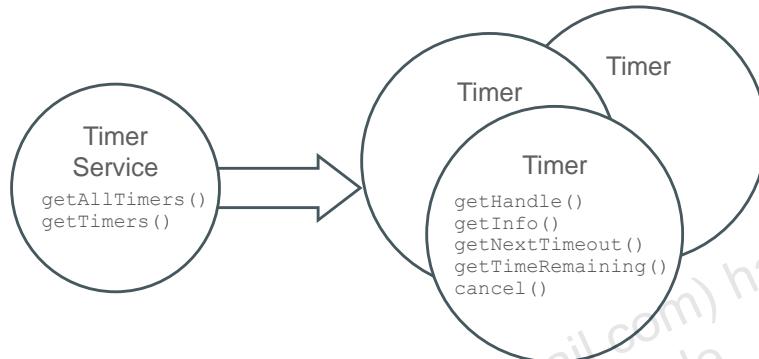
Alternatively, similar functionality to that one described on the page above, can be achieved by defining timer using the `ejb-jar.xml` file:

```
<ejb-jar>
    <enterprise-beans>
        <session>
            <ejb-name>SomeBean</ejb-name>
            <ejb-class>demos.SomeBean</ejb-class>
            <session-type>Stateless</session-type>
            <timer>
                <schedule>
                    <second>\*30</second>
                    <minute>\*.10</minute>
                    <hour>\*</hour>
                    <month>\*</month>
                    <year>\*</year>
                </schedule>
                <timeout-method>
                    <method-name>someMethod</method-name>
                    <method-params>
                        <method-param>javax.ejb.Timer</method-param>
                    </method-params>
                </timeout-method>
            </timer>
        </session>
    </enterprise-beans>
</ejb-jar>
```

Manage Timers

TimerService object can be used to retrieve active Timer objects. Each Timer object can be used to:

- Obtain its handler
- Obtain an info object associated with this timer
- Obtain information about its next timeout and remaining time
- Cancel Timer



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

You can retrieve active timers from the TimerService object:

- `getAllTimers`: Returns all active timers associated with any bean in the same ejb-module as the caller bean
- `getTimers`: Returns all active timers associated with this bean

Each Timer object allows:

- `getHandle`: Returns a serializable handle to the timer object. By persisting this handle, you can obtain a reference to the timer object at a future date.
- `getInfo`: Returns the info object associated with the timer during timer creation. The info object can contain any kind of information as long as it is serializable.
- `getNextTimeout`: Returns a date object that represents the absolute time until the next scheduled timer expiration
- `getTimeRemaining`: Returns a long value that represents the time in milliseconds until the next scheduled timer expiration
- `cancel`: To cancel a timer, you invoke the `cancel` method on the timer. You can perform this task in any method that contains a valid reference to a timer object. Only the bean that created the timer can invoke the `cancel` method. Invoking the `cancel` method removes the timer from the container. After a timer has been canceled, subsequent method invocations on the timer will generate `NoSuchObjectException`.

Define Interceptors

Interceptors allows you to separate reusable/repeated code fragments such as logging, auditing and security, from the business logic implementation code contained in CDI or EJB components.

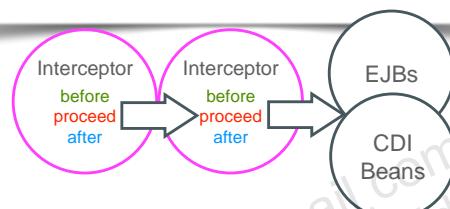
How interceptors are defined:

- A method in the target component class itself
- A separate class that can be applied to a number of CDI and EJB components

How Interceptors are applied:

- Caller invokes components as usual.
- Each interceptor applied to this component executes its "before" part.
- Then it passes control to the next interceptor and eventually to the target component.
- Then it executes the "after" part.
- Priority annotation can be used to set interceptor execution order

```
@Interceptor
@Priority(Interceptor.Priority.APPLICATION+1)
public class SomeInterceptor {
    @AroundInvoke
    public Object yourMethod(InvocationContext ctx) {
        // Perform "Before" Actions
        Object obj = ctx.proceed();
        // Perform "After" Actions
        return obj;
    }
    @AroundTimeout
    public Object methodD(InvocationContext ctx){...}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The AroundInvoke annotation defines an interceptor method that interposes on business methods. May be applied to any nonfinal, nonstatic method with a single parameter of type InvocationContext and return type Object of the target class (or superclass) or of any interceptor class.

Such operations can be placed in dedicated interceptor classes:

```
@Interceptor
public class SomeInterceptor {
    @AroundInvoke
    public Object yourMethod(InvocationContext ctx) {
        ...
        Object obj = ctx.proceed();
        ...
        return obj;
    }
}
```

Or be added directly to target component class:

```
@Stateless  
public class SomeBean {  
    @AroundInvoke  
    public Object yourMethod(InvocationContext ctx) {  
        ...  
        Object obj = ctx.proceed();  
        ...  
        return obj;  
    }  
    ...  
}
```

Priorities that define the order in which interceptors are invoked. These values should be used with the Priority annotation.

- Interceptors defined by platform specifications should have priority values in the range PLATFORM_BEFORE up until LIBRARY_BEFORE, or starting at PLATFORM_AFTER.
- Interceptors defined by extension libraries should have priority values in the range LIBRARY_BEFORE up until APPLICATION, or LIBRARY_AFTER up until PLATFORM_AFTER.
- Interceptors defined by applications should have priority values in the range APPLICATION up until LIBRARY_AFTER.

An interceptor that must be invoked before or after another defined interceptor can choose any appropriate value.

Interceptors with smaller priority values are called first. If more than one interceptor has the same priority, the relative order of these interceptors is undefined.

Types of Interceptors

Interceptors can be applied to business methods, timeout methods, and life-cycle operations:

- Business Method Interceptor:
 @AroundInvoke
- Timeout Interceptor:
 @AroundTimeout
- Lifecycle Interceptors:
 @AroundConstruct
 @PostConstruct
 @PreDestroy

```
@Interceptors(SomeInterceptor.class)
public class SomeBean {
    public SomeBean() {...}
    @PostConstruct
    public void init() {...}
    ...
}
```

```
@Interceptor
public class SomeInterceptor {
    @AroundConstruct
    public Object methodA(InvocationContext ctx) {
        // Perform "Before" Actions
        Object obj = ctx.proceed();
        // Perform "After" Actions
        return obj;
    }
    @PostConstruct
    public Object methodB(InvocationContext ctx){...}
    @PreDestroy
    public Object methodC(InvocationContext ctx){...}
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The AroundConstruct annotation designates an interceptor method that receives a callback when the target class constructor is invoked.

The method to which this annotation is applied must have one of the following signatures:

@AroundConstruct

```
public void <METHOD>(InvocationContext ctx) { ... }
```

@AroundConstruct

```
public Object <METHOD>(InvocationContext ctx) throws Exception { ... }
```

@PostConstruct and **PreDestroy** annotations define interceptor operations that are applied to the corresponding bean life-cycle events.

The AroundTimeout annotation defines an interceptor method that interposes on timeout methods. It may be applied to any nonfinal, nonstatic method with a single parameter of type `InvocationContext` and return type `Object` of the target class (or superclass) or of any interceptor class.

`InvocationContext.getTimer()` allows any AroundTimeout method to retrieve the timer object associated with the timeout.

Apply Interceptors

- An interceptor can be applied to multiple components.
- Multiple interceptors can be applied to the same target component.
 - A, B, C, and D interceptors are applied to operationX.
 - A and B interceptors are applied to operationY.
 - No interceptor is applied to operationZ.

```
@Stateless  
{@Interceptors(InterceptorA.class, InterceptorB.class)  
public class SomeBean {  
    {@Interceptors(InterceptorC.class, InterceptorD.class)  
    public void operationX() {...}  
    public void operationY() {...}  
    @ExcludeInterceptors  
    public void operationZ() {...}  
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Interceptor operations may be described directly in the target component class. In this case they are applied to all business operations of this component:

```
@Stateless  
public class SomeBean {  
    @AroundInvoke  
    public Object yourMethod(InvocationContext ctx) {  
        ...  
        Object obj = ctx.proceed();  
        ...  
        return obj;  
    }  
    ...  
    public void operationL() {...}  
    public void operationM() {...}  
}
```

Summary

In this lesson, you should have learned how to:

- Create Session EJB components
- Create EJB business methods
- Manage EJB life cycle with container callbacks
- Use asynchronous EJB operations
- Control transactions
- Create EJB timers
- Create and apply interceptors



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Practice Overview

This practice covers the following tasks:

- Create Product Facade Stateless Session EJB
- Add business methods to handle persistency with JPA components
- Expose EJB operations through remote interface
- Create Java Client Application that invokes the EJB
- Create ExpiringProduct Singleton Timer EJB
- Make ExpiringProduct Singleton EJB invoke ProductFacade EJB using local call



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Using Java Message Service API

5



ORACLE®

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Jairo Jose Silvera Sarmiento (jairo.silvera@gmail.com) has a
non-transferable license to use this Student Guide.

Objectives

After completing this lesson, you should be able to:

- Describe Java Message Service (JMS) API messaging models
- Implement Java SE and Java EE message producers and consumers
- Use durable and shared topic consumer subscriptions
- Create message-driven beans
- Use transactions with JMS



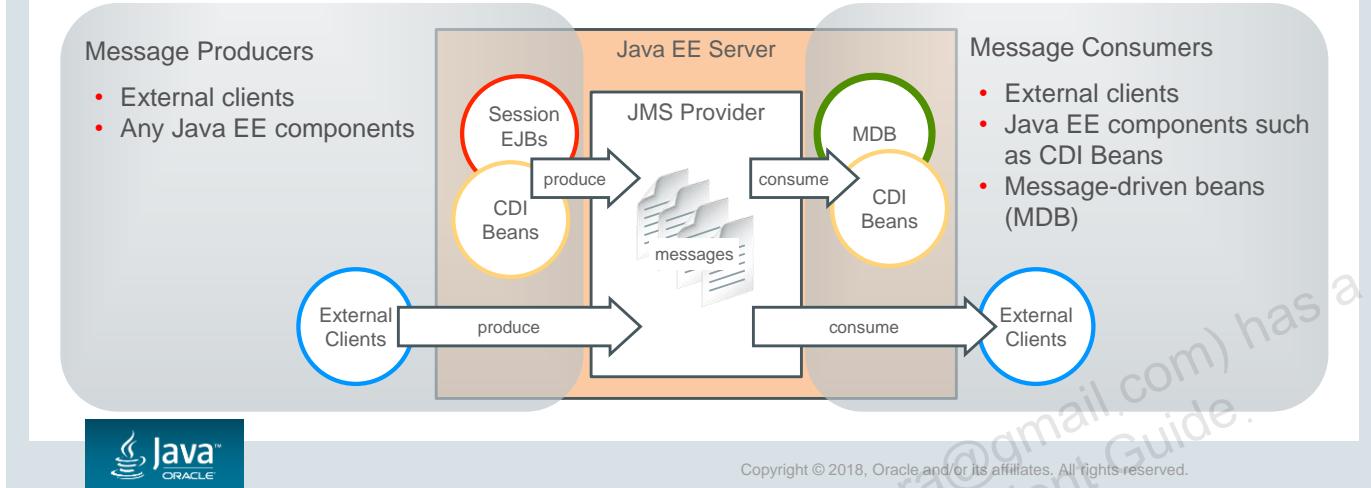
Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Java Message Service (JMS) API

JMS organizes message exchanges between producers and consumers via an intermediary.

- A JMS provider is a messaging system that implements the JMS interfaces and provides administrative and control features.
- An implementation of the Full Java EE platform includes a JMS provider.



The primary reason for using Java Message Service is to define a service for asynchronous request processing. Further, JMS provides a mechanism to allow multiple parties to retrieve messages based on subscription.

Despite the word Java in Java Message Service, the other benefit of message services is that there is no requirement that both the producers and consumers be written in the same language, or run on the same platform.

JMS represents a very loosely coupled type of interaction between components - generally producers and consumer don't have to be aware of each other.

A JMS application is composed of the following parts:

- **JMS provider:** A messaging system that implements the JMS API in addition to other administrative and control functionality required of a full-featured messaging product
- **JMS clients:** Programs that send and receive messages
- **Messages:** Objects that are used to communicate information between the clients of an application
- **Administered objects:** Provider-specific objects that clients look up and use to interact portably with a JMS provider

A Message-Driven Bean is a Java EE EJB component that is specifically designed to represent an Asynchronous Message Consumer:

- Can be subscribed to receive messages from a Queue or a Topic
- Can never be directly invoked by a client
- Typically, responsible for acquiring, validating, and preparing messages before passing them on to the Business Logic handling classes

Message-producer clients create and send messages to destinations.

Message-consumer clients consume messages from destinations. Message consumers can be either asynchronous or synchronous:

- 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.

JMS Destination Types

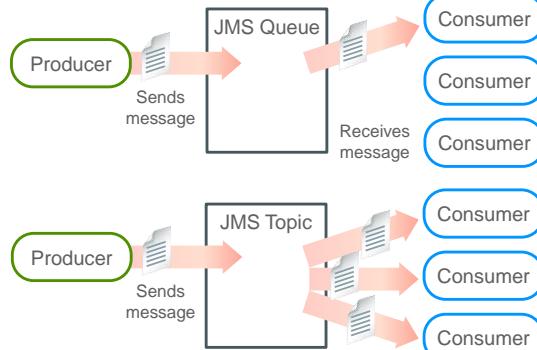
JMS Destinations are Queues and Topics through which messages are exchanged

Point-to-Point (Queue) Model

- Queue retains messages until they are consumed.
- Each message is consumed by one consumer.
- Messages may be "peeked at" by Queue Browser

Publish-Subscribe (Topic) Model

- Each message can have multiple consumers.
- Message may have no consumers, unless consumer uses durable subscription (identified by subscription name).
 - Durable subscriptions guarantee that all messages sent to the topic are received, even if there are no subscribers to the topic.
 - Nondurable subscriptions exist only for as long as there is an active consumer on the subscription. Any message sent to the topic when there are no subscribers is lost.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

JMS supports two styles of messaging:

- **Point-to-point (PTP):** Messaging by using queues
- **Publish/subscribe:** Messaging by using topics

In a point-to-point scenario, producers address messages to a queue. Point-to-point messaging has the following characteristics:

- A message queue retains all messages until they are consumed.
- There are no timing dependencies between the sender and receiver.
- A receiver gets all the messages that were sent to the queue—even those that were sent before the creation of the receiver.
- The queue then deletes the messages on acknowledgement of successful processing from the message consumer.
- 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. However, 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.

In a publish/subscribe scenario, producers address messages to a topic, which functions somewhat like a bulletin board. Publishers and subscribers can dynamically publish or subscribe to the topic. The system takes care of distributing the messages that arrive from a topic's multiple publishers to its multiple subscribers. Topics retain messages only as long as it takes to distribute them to subscribers.

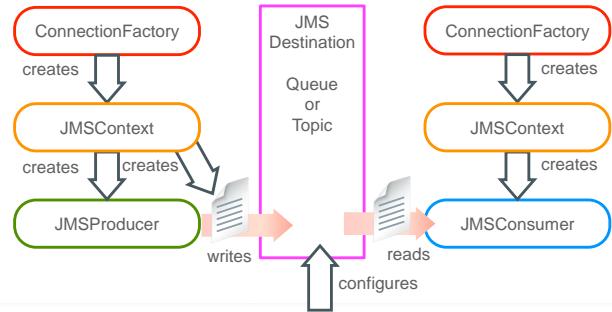
Publish/subscribe messaging has the following characteristics:

- Each message can have multiple consumers.
- A client that subscribes to a topic will only receive messages sent to that topic after the client creates the subscription. Messages sent to the topic before the client subscribes are lost. However, JMS API relaxes this requirement to some extent by allowing applications to create durable subscriptions, which receive messages sent while the consumers are not active.

JMS 2.0 API

JMS 2.0 API provides the following interfaces:

- **ConnectionFactory**: Creates JMSContext
- **JMSContext**: Represents active connected session
- **JMSPublisher**: Creates and writes messages
- **JMSPublisher**: Reads messages



Configure JMS Destinations

- JMS Server provider-specific Administration Tools
- Annotations
- Deployment descriptors

```

@JMSConnectionFactoryDefinition(
    name="java:global/jms/MyConnectionFactory",
    maxPoolSize=30, minPoolSize=20,
    properties={
        "addressList=t3://localhost:7001",
        "reconnectEnabled=true"})
@JMSDestinationDefinitions(
    {@JMSDestinationDefinition(name="java:global/jms/orderQueue",
        destinationName="orderQueue",
        interfaceName="javax.jms.Queue"})
public class OrderManagement {...}
  
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

JMS API provides following interfaces:

- **ConnectionFactory**: An administered object that is used by a client to create a JMSContext
- **JMSContext**: An active connection to a JMS provider and a single-threaded context for sending and receiving messages
- **JMSPublisher**: An object created by a JMSContext that is used for sending messages to a queue or topic
- **JMSPublisher**: An object created by a JMSContext that is used for receiving messages sent to a queue or topic

Administered objects are objects that are configured administratively (as opposed to programmatically) for each messaging application. A JMS provider supplies the administered objects.

There are two types of JMS-administered objects:

- **ConnectionFactory**: The object that a client uses to create a connection with a JMS provider
- **Destination**: The object that a client uses to specify the destination of the messages that it is sending and the source of the messages that it receives

JMS Connection Factories, Queues, and Topics can be configured via server administrative tools, annotations, and deployment descriptors.

This example defines an equivalent configuration to the one defined via the annotation above:

```
<jms-connection-factory>
    <name>java:global/jms/MyConnectionFactory</name>
    <max-pool-size>30</max-pool-size>
    <min-pool-size>20</min-pool-size>
    <property>
        <name>addressList</name>
        <value>t3://localhost:7001</value>
    </property>
    <property>
        <name>reconnectEnabled</name>
        <value>true</value>
    </property>
</jms-connection-factory>

<jms-destination>
    <name>java:global/jms/orderQueue</name>
    <interfaceName>javax.jms.Queue</interfaceName>
    <destinationName>orderQueue</destinationName>
</jms-destination>
```

JMS providers differ significantly in their implementations of the underlying messaging technology. There are also major differences in how a JMS provider's system is installed and administered.

For JMS clients to be portable, they must be isolated from the proprietary aspects of a provider. This is done by defining JMS-administered objects that are created and customized by a provider's administrator and later used by clients. The client uses them through JMS interfaces that are portable. The administrator creates them by using provider-specific facilities.

Administered objects are placed in a Java Naming and Directory Interface (JNDI) namespace by an administrator. A JMS client typically notes in its documentation the JMS-administered objects it requires and how the JNDI names of these objects should be provided to it.

The two types of administered objects are:

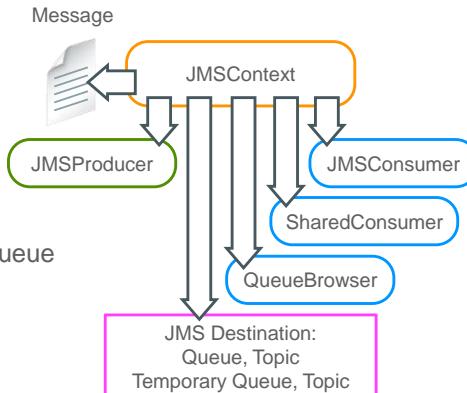
- Destinations: They are message distribution points. Destinations receive, hold, and distribute messages. Destinations fall into two categories: queue and topic destinations.
 - Queue destinations implement the point-to-point messaging protocol.
 - Topic destinations implement the publish/subscribe messaging protocol.
- Connection factories: They are used by a JMS API client (JMS client) to create a connection to a JMS API destination (JMS destination).

JMS Context

JMSContext Object represents connected JMS session.

It is used to:

- Create Byte, Map, Object, Stream, and Text messages
- Create JMS Producers, Consumers and Queue Browsers
 - JMS Producer: To post messages
 - JMS Consumer: To receive messages
 - Shared Consumer
 - Queue Browser - To peek at the messages on the specified queue
- Create JMS Destinations
 - Queue or Topic
 - Temporary Queue or Topic
- Set up Exception Handlers
- Control Bean Managed Transactions
- Control subscriptions and delivery of messages



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

JMSContext object provides a number of message creation methods:

- `createMessage()`
- `createBytesMessage()`
- `createMapMessage()`
- `createObjectMessage()`
- `createObjectMessage(Serializable object)`
- `createStreamMessage()`
- `createTextMessage()`
- `createTextMessage(String text)`

For more information on JMS Messages see:

<https://docs.oracle.com/javaee/7/api/javax/jms/Message.html>

JMSContext object creates JMS Producer objects:

- `createProducer()`

JMSContext object creates JMS Consumer objects:

- `createConsumer(Destination destination)`
- `createConsumer(Destination destination, String messageSelector)`
- `createConsumer(Destination destination, String messageSelector, boolean noLocal)`
- `createDurableConsumer(Topic topic, String subscriptionName)`
- `createDurableConsumer(Topic topic, String subscriptionName, String messageSelector, boolean noLocal)`

JMSContext object creates Shared Consumer objects:

- `createSharedConsumer(Topic topic, String sharedSubscriptionName)`
- `createSharedConsumer(Topic topic, String sharedSubscriptionName, String messageSelector)`
- `createSharedDurableConsumer(Topic topic, String sharedSubscriptionName)`
- `createSharedDurableConsumer(Topic topic, String sharedSubscriptionName, String messageSelector)`

JMSContext object creates QueueBrowser objects to peek at the messages on the specified queue:

- `createBrowser(Queue queue)`
- `createBrowser(Queue queue, String messageSelector)`

JMSContext object creates JMS Destinations:

- `createQueue(String queueName)`
- `createTopic(String topicName)`
- `createTemporaryQueue()`
- `createTemporaryTopic()`

For more information, see:

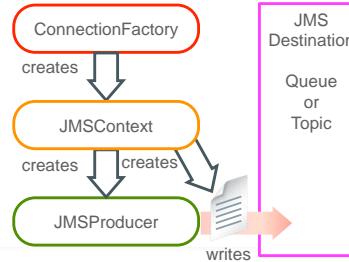
<https://docs.oracle.com/javaee/7/api/javax/jms/JMSContext.html>

Java SE Message Producer

Java SE Message Producer:

- Obtains JNDI Initial Context object
- Looks up **ConnectionFactory**, **Queue**, or **Topic** resources
- Creates **JMSContext** object
- Creates **JMSPublisher** object
- Optionally, sets producer properties (see notes)
- Prepares and sends messages

```
public class OrderProducer {
    public static void main(String[] args) {
        Context ctx = new InitialContext();
        ConnectionFactory cf = (ConnectionFactory)ctx.lookup("java:comp/DefaultJMSConnectionFactory");
        Queue queue = (Queue)ctx.lookup("java:global/jms/orderQueue");
        try(JMSContext context = cf.createContext()){
            JMSPublisher producer = context.createPublisher(queue);
            Message msg = JMSPublisher.createMessage();
            producer.setDeliveryMode(DeliveryMode.PERSISTENT).setTimeToLive(1000).send(msg);
        }catch(JMSException e) {...}
    }
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Java SE client require server-specific libraries in their classpath.

- Glassfish JMS client uses `gfclient.jar` and `app client.jar`
- WebLogic JMS client uses `wlthint3client.jar`

There are several overloaded versions of the `send` method available in the `JMSPublisher`:

`send(Destination destination, byte[] body)` : Sends a `BytesMessage`

`send(Destination destination, Map<String, Object> body)` : Sends a `MapMessage`

`send(Destination destination, Message message)` : Sends a `javax.jms.Message` object (it can contain headers and body parts, where headers contain values used by both clients and providers to identify and route messages and body could be a Stream, Map, Text, Object or Bytes content)

`send(Destination destination, Serializable body)` : Sends an `ObjectMessage`

`send(Destination destination, String body)` : Sends a `TextMessage`

You can set various standard as well as custom properties for the `JMSPublisher`, such as:

JMS supports two modes of message delivery.

- The `NON_PERSISTENT` mode is the lowest overhead delivery mode because it does not require that the message be logged to stable storage. A JMS provider failure can cause a `NON_PERSISTENT` message to be lost.
- The `PERSISTENT` mode instructs the JMS provider to take extra care to ensure the message is not lost in transit due to a JMS provider failure.

`setTimeToLive(long timeout)` is used to determine the expiration time of a message.

- Specifies the time to live of messages that are sent using this `JMSProducer`. Value is in milliseconds; a value of zero (default) means that a message never expires.
- The expiration time of a message is the sum of the message's time to live and the time it is sent. For transacted sends, this is the time the client sends the message, not the time the transaction is committed.
- Clients should not receive messages that have expired; however, JMS does not guarantee that this will not happen.
- A JMS provider should do its best to accurately expire messages; however, JMS does not define the accuracy provided. It is not acceptable to simply ignore time-to-live.

setDeliveryDelay(long deliveryDelay): Sets the minimum length of time in milliseconds that must elapse after a message is sent before the JMS provider may deliver the message to a consumer.

setPriority(int priority): Specifies the priority of messages that are sent using this `JMSProducer`

The JMS API defines 10 levels of priority value, with 0 as the lowest priority and 9 as the highest. Clients should consider priorities 0-4 as gradations of normal priority and priorities 5-9 as gradations of expedited priority. Priority is set to 4 by default.

You can also reduce message processing overhead by disabling Message Ids and Message Timestamps, in case your application does not need to use these properties to handle messages.

For more information see:

<https://docs.oracle.com/javaee/7/api/javax/jms/JMSProducer.html>

In the JMS 2.0 API `JMSProducer` and `JMSSConsumer` objects combine features that were handled by `Connection` and `Session` classes in the earlier JMS APIs such as JMS 1.1.

Example below shows JMS 1.1 message producer:

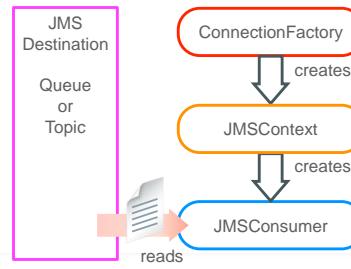
```
public class OrderProducer {
    public static void main(String[] args) {
        Context ctx = new InitialContext();
        ConnectionFactory cf =
        (ConnectionFactory) ctx.lookup("java:comp/DefaultJMSConnectionFactory");
        Queue queue = (Queue) ctx.lookup("java:global/jms/orderQueue");
        try (Connection connection = connectionFactory.createConnection();
             Session session =
             connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
             MessageProducer messageProducer =
             session.createProducer(queue)) {
            Message msg = session.createMessage();
            messageProducer.send(msg);
        } catch (JMSException ex) {
            ...
        }
    }
}
```

Java SE Message Consumer

Java SE Message Consumer:

- Obtains JNDI Initial Context object
- Looks up **ConnectionFactory**, **Queue**, or **Topic** resources
- Creates **JMSContext** object
- Creates **JMSConsumer** object
- Receives messages

```
public class OrderConsumer {
    public static void main(String[] args) {
        Context ctx = new InitialContext();
        ConnectionFactory cf = (ConnectionFactory) ctx.lookup("java:comp/DefaultJMSSConnectionFactory");
        Queue queue = (Queue) ctx.lookup("java:global/jms/orderQueue");
        try(JMSContext context = cf.createContext();
            JMSConsumer consumer = context.createConsumer(queue)) {
            Message message = consumer.receive(1000);
        } catch (JMSEException e) {...}
    }
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

JMS consumer provides several ways of receiving messages from JMS Queues and Topics:

- Method `receive()`: Receives the next message produced for this message consumer
 - This call blocks indefinitely until a message is produced or until this message consumer is closed.
 - If this receive is done within a transaction, the consumer retains the message until the transaction commits.
- Method `receive(long timeout)`: Receives the next message that arrives within the specified timeout interval
 - Parameters `timeout` sets the timeout value in milliseconds.
 - This call blocks until a message arrives, the timeout expires, or this message consumer is closed.
 - A timeout of zero never expires, and the call is blocked indefinitely.
- Method `receiveNoWait()`: Receives the next message if one is immediately available
 - It returns the next message produced for this `JMSConsumer`, or null if one is not available.

These operations return the next message produced for this message consumer, or null if this message consumer is concurrently closed.

They could throw `JMSEExceptions`, if the JMS provider fails to receive the next message due to some internal error.

JMSConsumer defines three equivalent overloaded operations called `receiveBody` that allow retrieving just the message body, without analyzing headers or properties of the message.

For more information, see:

<https://docs.oracle.com/javaee/7/api/javax/jms/JMSConsumer.html>

Java SE Asynchronous Producers and Consumers

Java SE JMS Producer may send messages asynchronously

- While JMSProducer handles the send operation, the program can perform other actions.
- CompletionListener operations will be invoked when the send operation finishes.

```
CompletionListener listener = new CompletionListener() {
    public void onCompletion(Message message) {...}
    public void onException(Message message, Exception exception) {...}
};
context.createProducer().setAsync(listener).send(jmsDestination, message);
```

Java SE JMS Consumer may receive messages asynchronously.

- While JMSConsumer handles the receive operation, the program can perform other actions.
- MessageListener onMessage operation will be invoked when the message arrives.

```
MessageListener listener = new MessageListener() {
    public void onMessage(Message message) {...}
};
context.createConsumer().setMessageListener(listener);
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Java SE application must prepare all JMS resources (ConnectionFactory, JMSContext, JMSProducer) in the same way no matter if it is going to use synchronous or asynchronous message delivery mode.

Sending messages asynchronously allows your application not to wait for the send method to return, confirming that the message has been delivered successfully, or throw an exception if there was a problem. This enables your application to perform other tasks, or send more messages, while given message delivery is still in-progress. In order to find out if the message has been delivered successfully or not you must override two methods defined by the javax.jms.CompletionListener

- onCompletion - this method notifies the application that the message has been successfully sent
- onException – this method notifies the application that the exception was thrown while attempting to send specified message

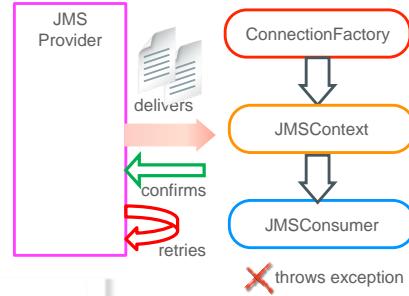
In order to prevent asynchronous JMS producer from getting into indefinite loop trying to redeliver a faulty message, you should also check how many times the message has been redelivered:

```
int deliveryCount = message.getIntProperty("JMSXDeliveryCount");
```

JMS Session Modes and Message Acknowledgments

JMSContext session modes:

- AUTO_ACKNOWLEDGE (default)
- DUPS_OK_ACKNOWLEDGE
- CLIENT_ACKNOWLEDGE
- SESSION_TRANSACTED



```

ConnectionFactory cf = ...
JMSContext context = cf.createContext(JMSContext.SESSION_TRANSACTED);
  
```

Message delivery can be confirmed explicitly, or in transacted session scenario messages are acknowledged implicitly with transaction commit.

Throwing runtime exception may cause message to be redelivered depending on the session mode (see notes).

```

context.acknowledge();
context.recover();

context.commit();
context.rollback();
  
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Message delivery can be acknowledged explicitly. However, for transacted sessions commit would implicitly acknowledge processed messages.

- acknowledge() - Acknowledges all messages consumed by the JMSContext's session.
- commit() - Commits all messages done in this transaction and releases any locks currently held.
- recover() - Stops message delivery in the JMSContext's session, and restarts message delivery with the oldest unacknowledged message.
- rollback() - Rolls back any messages done in this transaction and releases any locks currently held.

The result of a listener throwing a RuntimeException depends on the session's acknowledgment mode.

- AUTO_ACKNOWLEDGE or DUPS_OK_ACKNOWLEDGE - the message will be immediately redelivered. The number of times a JMS provider will redeliver the same message before giving up is provider-dependent. The JMSRedelivered message header field will be set, and the JMSXDeliveryCount message property incremented, for a message redelivered under these circumstances.
- DUPS_OK_ACKNOWLEDGE - instructs the JMSContext's session to lazily acknowledge the delivery of messages. This is likely to result in the delivery of some duplicate messages if the JMS provider fails, so it should only be used by consumers that can tolerate duplicate messages. Use of this mode can reduce session overhead by minimizing the work the session does to prevent duplicates.

- CLIENT_ACKNOWLEDGE - the next message for the listener is delivered. If a client wishes to have the previous unacknowledged message redelivered, it must manually recover the session.
- Transacted Session - the next message for the listener is delivered. The client can either commit or roll back the session (in other words, a RuntimeException does not automatically rollback the session).

JMS providers should flag clients with message listeners that are throwing RuntimeException as possibly malfunctioning.

Handle JMS Messages

JMS Message Producer

- Create Message of specific type:
 - ByteMessage
 - MapMessage
 - ObjectMessage
 - StreamMessage
 - TextMessage
- Set message properties

```
ObjectMessage msg = context.createObjectMessage(product);
msg.setStringProperty("name",product.getName());
msg.setFloatProperty("price",product.getPrice());
producer.send(orderQueue, msg);
```

JMS Message Consumer

- Retrieves properties
- Checks and casts message type to:
 - ByteMessage
 - MapMessage
 - ObjectMessage
 - StreamMessage
 - TextMessage

```
public void onMessage(Message msg) {
    try {
        String name = msg.getStringProperty("product");
        float price = msg.getFloatProperty("price");
        if (msg instanceof ObjectMessage) {
            ObjectMessage objMsg = (ObjectMessage)msg;
            Product product = (Product)objMsg.getObject();
        }
    }catch (Exception e){...}
}
```

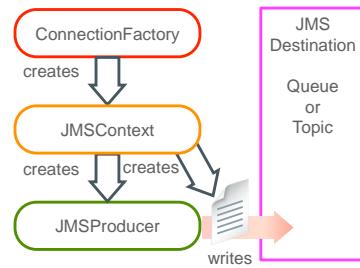


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Java EE Message Producer

Java EE Message Producer:

- May simply inject **JMSContext** object
 - **ConnectionFactory** can be automatically provided by container.
- Injects **Queue or Topic** Resource
- Creates **JMSPublisher** object
- Optionally, sets delivery properties and headers, such as delay
- Uses JMS Producer to send messages



```

public class OrderManagement {
    @Inject
    private JMSContext context;
    @Resource(lookup="jms/orderQueue");
    private Queue orderQueue;
    public void sendOrderMessage(Order order) {
        ObjectMessage msg = context.createObjectMessage(order);
        context.createProducer().setDeliveryDelay(20000).send(orderQueue, msg);
    }
}
  
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The `@Inject` annotation tells the container to create the `JMSContext` when it is needed.

Use `@Inject` to inject the `JMSContext`, specifying the connection factory that you want to use. The container automatically looks up this connection factory for you and uses it to create a `JMSContext`. Your code simply must use it. At the end of the transaction, the `JMSContext` is closed automatically for you by the container.

If you want to specify a connection factory, you do so via annotation:

```

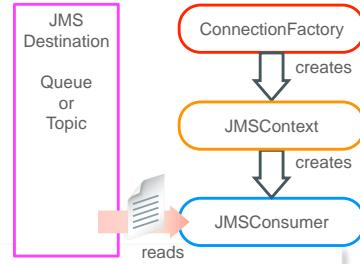
@Inject
@JMSConnectionFactory("java:comp/MyJMSConnectionFactory")
private JMSContext context;
  
```

Java EE Message Consumer

Java EE Message Consumer:

- May simply inject **JMSContext** object
 - **ConnectionFactory** can be automatically provided by container.
- Injects **Queue or Topic** Resource
- Creates **JMSConsumer** object
- Uses JMSConsumer to receive messages

```
@RequestScoped
public class InvoiceManagement {
    @Inject
    private JMSContext context;
    @Resource(literal="jms/orderQueue");
    private Queue orderQueue;
    public void receiveOrderMessage(){
        try(JMSConsumer consumer=context.createConsumer(orderQueue)){
            Message msg = consumer.receive(1000);
        }catch(JMSEException e){....}
    }
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

`createDurableConsumer(Topic topic, String name)`

Creates an unshared durable subscription on the specified topic (if one does not already exist) and creates a consumer on that durable subscription

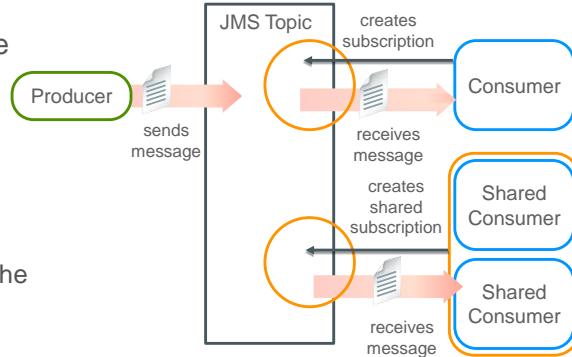
`JMSConsumer createDurableConsumer(Topic topic, String name, String messageSelector, boolean noLocal)`

Creates an unshared durable subscription on the specified topic (if one does not already exist), specifying a message selector and the `noLocal` parameter, and creates a consumer on that durable subscription

Topics Shared/Unshared Subscriptions

You can have many subscriptions on a topic. Each message is copied to every **subscription** (unless there is a message selector).

- Normal (Unshared) subscription represents a single consumer.
- If the subscription is shared, it can have many consumers. Each message from the **subscription** is delivered to only one of these consumers.
- A shared **subscription** is identified by a name (and by the client ID if it is set), and can have multiple consumers.



```
...
JMSConsumer consumer=context.createSharedConsumer(someTopic,"SomeSubscriptionName");
...
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

JMSContext object creates Shared Durable or Non-Durable Topic Consumer objects, with or without message selectors:

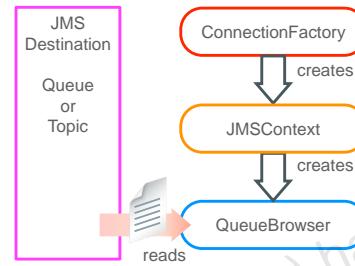
- `createSharedConsumer(Topic topic, String sharedSubscriptionName)`
- `createSharedConsumer(Topic topic, String sharedSubscriptionName, String messageSelector)`
- `createSharedDurableConsumer(Topic topic, String sharedSubscriptionName)`
- `createSharedDurableConsumer(Topic topic, String sharedSubscriptionName, String messageSelector)`

Queue Message Browser

Java EE Queue Message Browser:

- Injects **JMSContext** object
- Injects **Queue** Resource
- Creates **QueueBrowser** object
- Uses QueueBrowser to look at queue messages (they stay in the Queue)

```
@RequestScoped
public class InvoiceManagement {
    @Inject
    private JMSContext context;
    @Resource(lookup="jms/orderQueue");
    private Queue orderQueue;
    public void browseOrderMessages() {
        try(QueueBrowser browser = context.createBrowser(queue)) {
            Enumeration<Message> msgs = browser.getEnumeration();
            while (msgs.hasMoreElements()) {
                Message msg = msgs.nextElement();
                ...
            }
        } catch(JMSException e) { ... }
    }
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

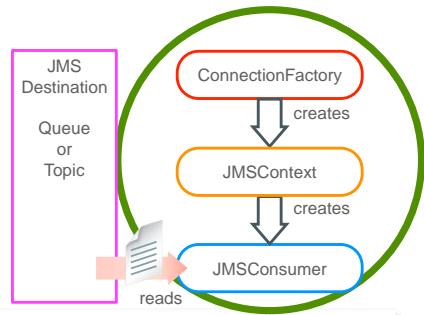
Java SE Queue Message browser example:

```
public class OrderProducer {
    public static void main(String[] args) {
        Context ctx = new InitialContext();
        ConnectionFactory cf =
        (ConnectionFactory)ctx.lookup("java:comp/DefaultJMSConnectionFactory");
        Queue queue = (Queue)ctx.lookup("java:global/jms/orderQueue");
        try (JMSContext context = connFactory.createContext();
             QueueBrowser browser = context.createBrowser(queue)) {
            Enumeration msgs = browser.getEnumeration();
            while (msgs.hasMoreElements()) {
                Message tempMsg = (Message) msgs.nextElement();
                ...
            }
        } catch (JMSException ex) { ... }
    }
}
```

Message-Driven Bean (MDB)

MDB is an asynchronous Java EE message consumer.

- Defined with `@MessageDriven` annotation, or using `ejb-jar.xml`
- Implements `MessageListener` interface
- Container manages connectivity, subscription, and message delivery for the MDB based on configuration provided
- Optionally, injects a `MessageDrivenContext` object in order to:
 - Access Security Properties
 - Control Transactions



```

@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName="destinationLookup",
        propertyValue="jms/productQueue"),
    @ActivationConfigProperty(propertyName="destinationType",
        propertyValue = "javax.jms.Queue"),
    @ActivationConfigProperty(propertyName="messageSelector",
        propertyValue="product='tea' AND price > 2.99"))
public class ProductMessageHandler implements MessageListener {
    @Inject
    private MessageDrivenContext context;
    public void onMessage(Message message) {...}
}
  
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Message-Driven bean requirements:

- MBD class must be public but not final or abstract.
- Annotate the message-driven bean class by using the `MessageDriven` metadata annotation.
- Optionally, use resource injection to obtain a `MessageDrivenContext` instance.
- Include a public, no-argument, empty constructor.
- Provide the `onMessage` method to consume messages.
- Do not define a `finalize` method.

Message-driven beans do not:

- Have remote component or local component interfaces
- 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.
- Have a client-visible identity. They do not hold client conversational states. They are anonymous to clients.
- 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.

@ActivationConfigProperty

The following standard properties are recognized for JMS message-driven beans:

- acknowledgeMode Auto_acknowledge (default) OR Dups_ok_acknowledge: This property is only used with Bean Managed Transactions. In Container Managed Transaction Scenario (default), container is responsible for message acknowledgement.
- messageSelector: Specifies which messages to receive using SQL-like syntax
 - If a message contains properties that match the selection criteria, it is delivered to the message-driven bean.
 - If the message does not match the criteria, the message-driven bean is not notified.
 - You can declare the JMS message selector by using the activation configuration property messageSelector.
- destinationType javax.jms.Queue or javax.jms.Topic
- destinationLookup: Specifies the queue or topic JNDI name
- connectionFactoryLookup: Specifies the JMS connection factory JNDI name
- subscriptionDurability Durable or NonDurable: Is used if the message-driven bean is intended to be used with a Topic, and the bean provider has indicated that a durable subscription should be used
- subscriptionName Specifies the name of the durable subscription
- clientId: Specifies the JMS client of the provider

JMS message-driven beans:

- They are annotated with @javax.ejb.MessageDriven.
- They implement the javax.jms.MessageListener interface.
- The application server can use integrated JMS support or use a connector.

Non-JMS message-driven beans:

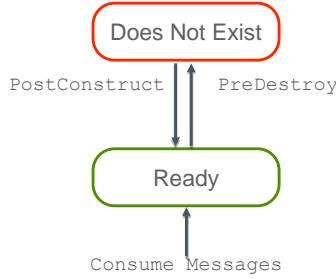
- They are annotated with @javax.ejb.MessageDriven.
- They implement a message listener interface specific to the messaging service.
- The application server requires a connector.

MDB Life Cycle

Message-Driven Bean lifecycle container callback operations:

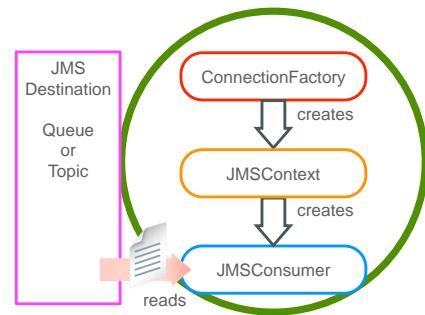
- PostConstruct is invoked when bean instance is created.
- PreDestroy is invoked when bean instance is disposed.

Message-Driven EJB Life Cycle



```

package demos;
// imports ...
@MessageDriven(...)
public class ProductMessageHandler implements MessageListener {
    @PostConstruct
    public void init() {...}
    @PreDestroy
    public void cleanup() {...}
    public void onMessage(Message message) {...}
}
  
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Containers can create and pool multiple instances of message-driven beans to service the same message destination.

The operation of these instances is independent of each other, that is, Message-Driven Beans are stateless.

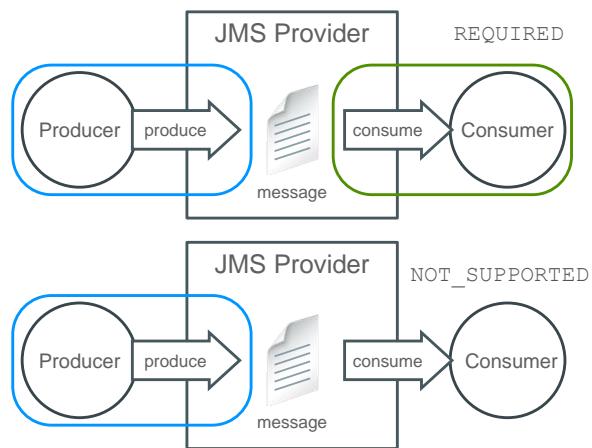
When a message arrives at a message destination, the container chooses the next available instance associated with that destination to service the message.

JMS and Transactions

The producer and consumer of messages do not share Security or Transaction Context.

- MDB can only use REQUIRED or NOT_SUPPORTED CMT Transactional semantics.
- For Transactional MDB, consider controlling number of delivery retries.

```
@MessageDriven(activationConfig = {
...
@ActivationConfigProperty(
    propertyName="messageSelector",
    propertyValue="JMSXDeliveryCount < 3") })
@TransactionalManagement(CONTAINER)
public class ProductMessageHandler implements MessageListener {
    @TransactionAttribute(REQUIRED)
    public void onMessage(Message msg) {
        ...
    }
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Because no client provides a transaction context for calls to a message-driven bean, beans that use container-managed transactions must be deployed using the Required or NotSupported transaction attribute.

When using the REQUIRED Transactional Attribute and if a container rolls back an MDB transaction, message consumption is rolled back and it can be retried. In this scenario, consider using JMSRedelivered and JMSXDeliveryCount properties to avoid getting into an indefinite loop of retrying same message delivery. To achieve this create another MDB that can be mapped to the same destination to handle errors that exceed the number of retries.

propertyValue="JMSXDeliveryCount > 3"

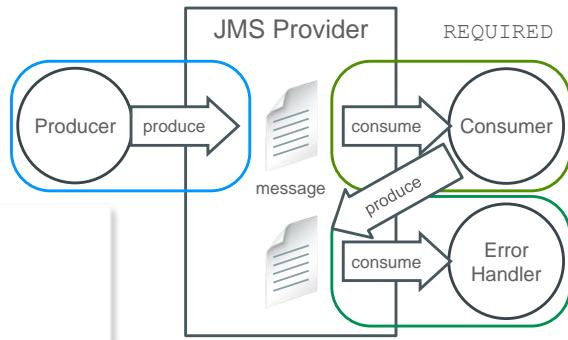


Handle Errors with Transactions

Transactional JMS Consumer

- It can put messages into "error queue" after exiting a number of retry attempts.
- Error Handler consumer can now take care of these messages.

```
@MessageDriven(...)
@TransactionManagement(CONTAINER)
public class ProductMessageHandler
    implements MessageListener {
    @TransactionAttribute(REQUIRED)
    public void onMessage(Message msg) {
        if (msg.getBooleanProperty("JMSRedelivered")) {
            int count = msg.getIntProperty("JMSXDeliveryCount");
            if (count > 3) {
                // move message away to "error queue"
            }
        }
    }
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

An Error Handler is just another JMS consumer subscribed to the "Error Queue" where you placed those messages that exceeded a number of retry attempts.

Summary

In this lesson, you should have learned how to:

- Describe Java Message Service (JMS) API messaging models
- Implement Java SE and Java EE message producers and consumers
- Use durable and shared topic consumer subscriptions
- Create message-driven beans
- Use transactions with JMS



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Practice Overview

This practice covers the following tasks:

- Create WebLogic JMS Server configuration
- Modify ExpiringProduct Timer EJB to produce messages
 - Post messages into the Queue
- Create Message-Driven Bean mapped to a Queue
 - Consume messages from the Queue
 - Pass messages to the Stateless EJB Facade business method



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Jairo Jose Silvera Sarmiento (jairo.silvera@gmail.com) has a
non-transferable license to use this Student Guide.

Implementing SOAP Services by Using JAX-WS

6



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



ORACLE®

Objectives

After completing this lesson, you should be able to handle communications with SOAP Services, including how to:

- Describe a SOAP Web Service structure
- Create SOAP Web Services using JAX-WS API
- Create SOAP Web Service clients



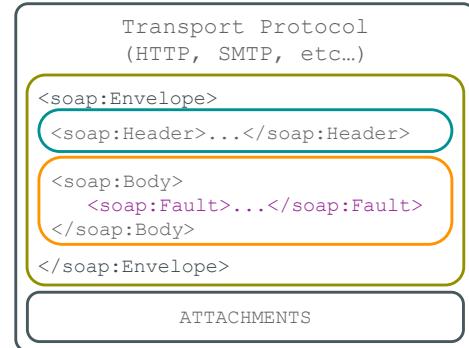
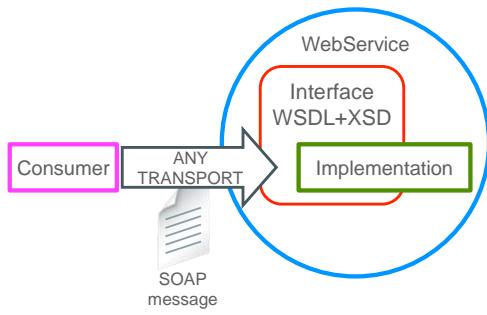
Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



WebServices and SOAP

SOAP WebServices

- **Web Services** are interoperable, platform independent mechanism for component interactions.
- SOAP decouples message representation from transport.
- **Service Interface** is described with WSDL and XSD files.
- **Service Implementation** are disguised — **Consumer** is unaware of implementation details of the service.



SOAP Message

- Can be transported over different protocols
- Must be enclosed into **SOAP Envelope**
- May have **Headers**, such as security, reliability, and so on
- Must have a **Body** with business data, or **Fault** elements
- May have attachments

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

There are two implementation styles of SOAP communications - RPC and Document.

- An RPC style service produces SOAP messages that contain the name of the service operation and its parameters.
- A Document style service simply relies on XML Schema to describe transmitted SOAP message body.

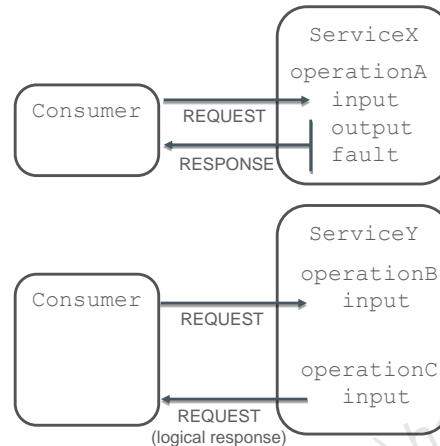
Document style is a recommended approach and RPC style is supported for backward compatibility reasons.

All examples in this course use Document style services.

Web Service Interaction Patterns

Web Services may use different interaction patterns.

- **Synchronous Service Operations:**
 - Describe Input, Output and optionally Fault parts
 - Consumer that invokes such operation sends a request containing the input and expects either fault to output response to be returned from the service.
 - Consumer is blocked for the duration of the call.
- **Asynchronous Service Operations:**
 - One-Way operations accept request from consumer and do not produce any response.
 - Two-Way operations accept request from consumer and may perform callback sending another request back to the original consumer.
 - Two-Way SOAP operations can be coordinated with WS-Addressing standard that describes callback addresses and correlations.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



A `@javax.jws.Oneway` annotation is used to map One-Way service operation to a Java method. It means that the operation has only input, and no output or fault is expected to be returned. Therefore, a Java method that implements this WebService operation should not have a return value, or Holder parameters or throw checked exceptions.

Web Service Interface

Web Services Description Language (WSDL) is an XML-based standard for describing SOAP Web Services interfaces.

- XML Schema Definition (XSD) documents describe data structures that services can use.
- WSDL file describes the service with:
 - **Abstract WSDL part:**
 - References XSD file to use data-structures it defines
 - Describes messages comprising elements from the XSD that this service would accept and return in its operations
 - Describes port types - logical groups of operations
 - Describes operations that would use messages to produce input, output, or fault parts
 - **Concrete WSDL part:**
 - Describes bindings to define how to invoke operations
 - Describes service to define an end-point invocation address where this service can be found

ProductQuote.xsd

```
<element>...</element>
<element>...</element>
```

ProductQuoteService.wsdl

```
<types>import xsd</types>
<message>...</message>
<message>...</message>
<portType>
  <operation>...</operation>
  <operation>...</operation>
</portType>
```

```
<binding>...</binding>
<service>...</service>
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

XML Schema Definition

XML Schema Definition document describes data-structures that web services can use.

- Define with a **unique namespace**.
- It describes a number of elements, types, and so on.
- **Components** described by the schema can be mapped to Java Classes using **Java Architecture for XML Binding (JAXB) API** with annotations or XML descriptors.

❖ JAXB API is covered in the Appendix E

```
<xs:schema version="1.0"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://xmlns.oracle.com/demos/ProductQuote"
  elementFormDefault="qualified">
  <xs:element name="Product">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="id" type="xs:int"/>
        <xs:element name="quantity" type="xs:int"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  ...
</xs:schema>
```

`@XmlAccessorType(XmlAccessType.FIELD)
@XmlElement(name = "", propOrder = {
 "id",
 "quantity"
})
@XmlRootElement(name = "Product")
public class Product {
 protected int id;
 protected int quantity;
 ...
}`



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

An example of the XML Schema definition file that models data structures for future use of the web service request, response, and fault messages:

```
<xs:schema version="1.0"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://xmlns.oracle.com/demos/ProductQuote"
  elementFormDefault="qualified">
  <xs:element name="Product">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="id" type="xs:int"/>
        <xs:element name="quantity" type="xs:int"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="Quote" type="xs:float"/>
  <xs:element name="QuoteError" type="xs:string"/>
</xs:schema>
```

Java class that is mapped to this xml schema Product element using JAXB annotations:

```
package com.oracle.xmlns.demos.productquote;  
import javax.xml.bind.annotation.XmlAccessType;  
import javax.xml.bind.annotation.XmlAccessorType;  
import javax.xml.bind.annotation.XmlRootElement;  
import javax.xml.bind.annotation.XmlType;  
  
@XmlAccessorType(XmlAccessType.FIELD)  
@XmlType(name = "", propOrder = {  
    "id",  
    "quantity"  
})  
@XmlRootElement(name = "Product")  
public class Product {  
    protected int id;  
    protected int quantity;  
    public int getId() {  
        return id;  
    }  
    public void setId(int value) {  
        this.id = value;  
    }  
    public int getQuantity() {  
        return quantity;  
    }  
    public void setQuantity(int value) {  
        this.quantity = value;  
    }  
}
```

WSDL Schemas and Namespaces

Web Services Description Language (WSDL) namespace definitions

- References XSD via its **unique namespace** to access data structures described by the XML Schema
- Defines its own **unique namespace**, for referencing components described within this WSDL file
- Each namespace can be assigned an arbitrary prefix **tns ns1** and so on

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="PriceQuote" targetNamespace="http://xmlns.oracle.com/demos/PriceQuoteService"
    xmlns:tns="http://xmlns.oracle.com/demos/PriceQuoteService"
    xmlns:ns1="http://xmlns.oracle.com/demos/ProductQuote"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap12/"
    xmlns="http://schemas.xmlsoap.org/wsdl/">
    <types>
        <xsd:schema>
            <xsd:import namespace="http://xmlns.oracle.com/demos/ProductQuote"
                schemaLocation="ProductQuote.xsd"/>
        </xsd:schema>
    </types>
    ...
</definitions>
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

WSDL Messages, PortTypes, and Operations

Abstract WSDL describes capabilities of the service:

- **Messages**, which comprise parts referencing elements described by XSDs
- **Port Types**, which define logical groups of operations
- **Operations**, which define:
 - input
 - output
 - faultreferencing previously defined messages

```
<?xml version="1.0"?>
<definitions ...>
  <types>...</types>
  <message name="QuoteRequest">
    <part name="request" element="ns1:Product"/>
  </message>
  <message name="QuoteResponse">
    <part name="response" element="ns1:Quote"/>
  </message>
  <message name="QuoteFault">
    <part name="response" element="ns1:QuoteError"/>
  </message>
  <portType name="PriceQuote">
    <operation name="getQuote">
      <input message="tns:QuoteRequest"/>
      <output message="tns:QuoteResponse"/>
      <fault name="QuoteFault"
            message="tns:QuoteFault"/>
    </operation>
  </portType>
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

WSDL Bindings and Services

Concrete WSDL describes access to the service:

- Bindings, which map port types and operations they define to transport protocol
- Service element, which describes locations where ports were deployed

```
<definitions ...>
...
<binding name="PriceQuoteSOAP" type="tns:PriceQuote">
    <soap:binding style="document"
                  transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="getQuote">
        <input>
            <soap:body use="literal" parts="request"/>
        </input>
        <output>
            <soap:body use="literal" parts="response"/>
        </output>
        <fault name="QuoteFault">
            <soap:fault use="literal" name="QuoteFault"/>
        </fault>
    </operation>
</binding>
<service name="PriceQuote">
    <port name="PriceQuote" binding="tns:PriceQuoteSOAP">
        <soap:address location="http://localhost:7001/demos/PriceQuote"/>
    </port>
</service>
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Here is an example of the Web Service Description file that models data structures for future use of the web service request, response, and fault messages:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="PriceQuote"
targetNamespace="http://xmlns.oracle.com/demos/PriceQuoteService"
            xmlns:tns="http://xmlns.oracle.com/demos/PriceQuoteService"
            xmlns:ns1="http://xmlns.oracle.com/demos/ProductQuote"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap12/"
            xmlns="http://schemas.xmlsoap.org/wsdl/">

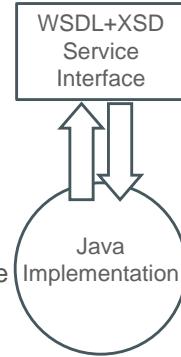
<types>
    <xsd:schema>
        <xsd:import
            namespace="http://xmlns.oracle.com/demos/ProductQuote"
            schemaLocation="ProductQuote.xsd"/>
    </xsd:schema>
</types>
<message name="QuoteRequest">
    <part name="request" element="ns1:Product"/>
</message>
```

```
<message name="QuoteResponse">
    <part name="response" element="ns1:Quote"/>
</message>
<message name="QuoteFault">
    <part name="response" element="ns1:QuoteError"/>
</message>
<portType name="PriceQuote">
    <operation name="getQuote">
        <input message="tns:QuoteRequest"/>
        <output message="tns:QuoteResponse"/>
        <fault name="QuoteFault" message="tns:QuoteFault"/>
    </operation>
</portType>
<binding name="PriceQuoteSOAP" type="tns:PriceQuote">
    <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="getQuote">
        <input>
            <soap:body use="literal" parts="request"/>
        </input>
        <output>
            <soap:body use="literal" parts="response"/>
        </output>
        <fault name="QuoteFault">
            <soap:fault name="QuoteFault" use="literal"/>
        </fault>
    </operation>
</binding>
<service name="PriceQuote">
    <port name="PriceQuote" binding="tns:PriceQuoteSOAP">
        <soap:address
location="http://localhost:7001/demos/PriceQuote"/>
    </port>
</service>
</definitions>
```

Top-down versus Bottom-up approach

Top-down approach (contract first)

- Start Web Service development by defining service interface using WSDL and XSD files
- Then create Java classes mapped to this interface definition
- Java service implementation classes can be auto-generated by the IDE
- Provides better quality Web Service interface description



Bottom-up approach (implementation first)

- Start Web Service development by defining Java classes that will implement the service
- Then create Web Service interface definition artefacts (WSDL and XSD files)
- WebService interface definition files can be auto-generated upon deployment
- Quicker development



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

There are utilities that automate a process of WebService creation, for either approach. For example:

- With top-down approach you can use a utility called `wsimport` that can generate Java implementation code based on WSDL and XSD files provided.
- With bottom-up approach you can use a utility called `wsgen` that can generate WSDL and XSD files based on Java implementation code provided.

Map Java Interface to WSDL

Map WSDL operations to java Interface using Java for XML Web Services (JAX-WS) API.

- Define **Java Interface** mapped to the **Web Service** described by WSDL file.
- Define **SOAP protocol binding**.
- Map **Java operations** to **WSDL operations** with the **WebMethod** annotation.
- Describe mappings for operation **return type** and **parameters**.
- Declare **exceptions** mapped to fault elements of the service operation.

```
...
@WebService(name="PriceQuote",targetNamespace="http://xmlns.oracle.com/demos/PriceQuoteService")
@SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
@XmlSeeAlso({ObjectFactory.class})
public interface PriceQuote {
    @WebMethod
    @WebResult(name="Quote",
               targetNamespace="http://xmlns.oracle.com/demos/ProductQuote",
               partName="response")
    public float getQuote(@WebParam(name = "Product",
                                    targetNamespace = "http://xmlns.oracle.com/demos/ProductQuote",
                                    partName = "request")Product request) throws QuoteFault;
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

JAX-WS Requirements for Web Service Methods

- Must be public. By default, every public method in the class will be a part of the web service.
- Must not be static or final
- Must have JAXB-compatible parameters and return types. Parameters and return types must not implement the `java.rmi.Remote` interface.

`WebMethod` annotations used to map Java method to WSDL operation. It has following properties:

- `operationName`: Name of the wsdl:operation matching this method
- `exclude`: Marks a method to NOT be exposed as a web method
- `action`: Associate this operation with the SOAP Action property. Unique label for this operation, that may optionally be passed as a SOAP Header to link client call to specific operation.

The `SOAPBinding` annotation defines parameter style mapping for the web service. It could have two values:

- `BARE`: Method parameters represent the entire message body.
- `WRAPPED`: Parameters are elements wrapped inside a top-level element named after the operation.

The `WebResult` annotation defines the datatype mapping between the return type of the Java operation and output message used by the operation described within the wsdl file.

The `WebParam` annotation defines mapping between parameters of the Java Method and input message used by the operation described within the wsdl file.

XMLSeeAlso annotation instructs JAXB API to bind other classes (in this case ObjectFactory.class) when binding this (PriceQuote.class).

ObjectFactory class contains helper methods to produce java objects (Product, Quote, QuoteFault) mapped to input, output, and fault elements used by operation getQuote that this interface maps for the service implementation.

Here is a complete interface code example:

```
package com.oracle.xmlns.demos.pricequotesservice;
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import javax.xml.bind.annotation.XmlSeeAlso;
import com.oracle.xmlns.demos.productquote.ObjectFactory;
import com.oracle.xmlns.demos.productquote.Product;
@WebService(name = "PriceQuote", targetNamespace =
"http://xmlns.oracle.com/demos/PriceQuoteService")
@SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
@XmlSeeAlso({
    ObjectFactory.class
})
public interface PriceQuote {
    @WebMethod
    @WebResult(name = "Quote",
        targetNamespace =
"http://xmlns.oracle.com/demos/ProductQuote",
        partName = "response")
    public float getQuote(
        @WebParam(name = "Product",
            targetNamespace =
"http://xmlns.oracle.com/demos/ProductQuote",
            partName = "request")
        Product request) throws QuoteFault;
}
```

JAX-WS Implementation

Create Java implementations of SOAP Services using JAX-WS API.

- Define **Java class** that will provide **Web Service** implementation described by the WSDL file.
- Reference previously described **Java interface** that mapped service operations.
- Describe protocol **Binding Type** (in this case, SOAP1.2 over HTTP).
- Define **Web Methods Implementations** for operations declared by the **Web Service Java Interface**.

```
...
@WebService(serviceName="PriceQuote",
    portName="PriceQuote",
    endpointInterface="com.oracle.xmlns.demos.pricequotesservice.PriceQuote",
    targetNamespace="http://xmlns.oracle.com/demos/PriceQuoteService",
    wsdlLocation="WEB-INF/wsdl/PriceQuote/PriceQuoteService.wsdl")
@BindingType(value="http://java.sun.com/xml/ns/jaxws/2003/05/soap/bindings/HTTP/")
public class PriceQuote {
    public float getQuote(Product request) throws QuoteFault {
        ...
    }
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The BindingType annotation is used to specify the binding to use for a web service endpoint implementation class. Default is SOAP1.1/HTTP. This example uses SOAP1.2/HTTP binding type.

```
package demos.ws;
import com.oracle.xmlns.demos.pricequotesservice.QuoteFault;
import com.oracle.xmlns.demos.productquote.Product;
import javax.jws.WebService;
import javax.xml.ws.BindingType;
@WebService(serviceName = "PriceQuote",
    portName = "PriceQuote",
    endpointInterface =
"com.oracle.xmlns.demos.pricequotesservice.PriceQuote",
    targetNamespace =
"http://xmlns.oracle.com/demos/PriceQuoteService",
    wsdlLocation = "WEB-
INF/wsdl/PriceQuote/PriceQuoteService.wsdl")
@BindingType(value =
"http://java.sun.com/xml/ns/jaxws/2003/05/soap/bindings/HTTP/")
public class PriceQuote {
    public float getQuote(Product request) throws QuoteFault {
        ...
    }
}
```

Create JAVA JAX-WS Client

JAX-WS API is used to implement both SOAP WebServices and Clients.

- Define **Java class** that will represent **Web Service Client**.
- Define **Web Service Port Type** references.

```
...
@WebServiceClient(name="PriceQuote",
                  targetNamespace="http://xmlns.oracle.com/demos/PriceQuoteService",
                  wsdlLocation="file:wsdl/PriceQuoteService.wsdl")
public class PriceQuote_Service extends Service {
    ...
    @WebEndpoint(name="PriceQuote")
    public PriceQuote getPriceQuote() {
        return super.getPort(
            new QName("http://xmlns.oracle.com/demos/PriceQuoteService", "PriceQuote"),
            PriceQuote.class);
    }
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Complete example of a Java client class that represents PriceQuote Web Service:

```
package com.oracle.xmlns.demos.pricequotesservice;

import java.net.MalformedURLException;
import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import javax.xml.ws.WebEndpoint;
import javax.xml.ws.WebServiceClient;
import javax.xml.ws.WebServiceException;
import javax.xml.ws.WebServiceFeature;
@WebServiceClient(name = "PriceQuote", targetNamespace =
"http://xmlns.oracle.com/demos/PriceQuoteService", wsdlLocation =
"file:wsdl/PriceQuoteService.wsdl")

public class PriceQuote_Service extends Service {
    private final static URL PRICEQUOTE_WSDL_LOCATION;
    private final static WebServiceException PRICEQUOTE_EXCEPTION;
    private final static QName PRICEQUOTE_QNAME = new
```

```

QName("http://xmlns.oracle.com/demos/PriceQuoteService", "PriceQuote");
static {
    URL url = null;
    WebServiceException e = null;
    try {
        url = new URL("file:wsdl/PriceQuoteService.wsdl");
    } catch (MalformedURLException ex) {
        e = new WebServiceException(ex);
    }
    PRICEQUOTE_WSDL_LOCATION = url;
    PRICEQUOTE_EXCEPTION = e;
}
public PriceQuote_Service() {
    super(__getWsdlLocation(), PRICEQUOTE_QNAME);
}
public PriceQuote_Service(WebServiceFeature... features) {
    super(__getWsdlLocation(), PRICEQUOTE_QNAME, features);
}
public PriceQuote_Service(URL wsdlLocation) {
    super(wsdlLocation, PRICEQUOTE_QNAME);
}
public PriceQuote_Service(URL wsdlLocation, WebServiceFeature... features) {
    super(wsdlLocation, PRICEQUOTE_QNAME, features);
}
public PriceQuote_Service(URL wsdlLocation, QName serviceName) {
    super(wsdlLocation, serviceName);
}
public PriceQuote_Service(URL wsdlLocation, QName serviceName,
WebServiceFeature... features) {
    super(wsdlLocation, serviceName, features);
}
@WebEndpoint(name = "PriceQuote")
public PriceQuote getPriceQuote() {
    return super.getPort(new
QName("http://xmlns.oracle.com/demos/PriceQuoteService", "PriceQuote"),
PriceQuote.class);
}

```

```
@WebEndpoint(name = "PriceQuote")
public PriceQuote getPriceQuote(WebServiceFeature... features) {
    return super.getPort(new
QName("http://xmlns.oracle.com/demos/PriceQuoteService", "PriceQuote"),
PriceQuote.class, features);
}
private static URL __getWsdlLocation() {
    if (PRICEQUOTE_EXCEPTION!= null) {
        throw PRICEQUOTE_EXCEPTION;
    }
    return PRICEQUOTE_WSDL_LOCATION;
}
}
```

Invoke SOAP Service from Java SE Client

Invoke Web Service from Java SE client.

- Acquire **Web Service Port**.
- Prepare **input parameters**.
- Invoke service **operations**.
- Handle **output values** and possible **faults**.

```
...
public static void main(String[] args) {
    PriceQuote priceQuotePort = new PriceQuote_Service().getPriceQuote();
    Product product = new Product();
    product.setId(1);
    product.setQuantity(4);
    try {
        float quote = priceQuotePort.getQuote(product);
    }catch(QuoteFault ex) {...}
}
...
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Invoke SOAP Service from Java EE Component

Invoke Web Service from Java EE Container Managed component such as Servlet or EJB.

- Inject **Web Service Reference** and acquire **Web Service Port**.
- Prepare **input parameters**.
- Invoke **service operations**.
- Handle **output values**.
- Handle possible **faults**.

```
...
@Stateless
public class OrderManagement {
    @WebServiceRef(
        wsdlLocation="http://localhost:7001/demos/PriceQuote?WSDL")
    private PriceQuote_Service service;
    public float getQuote(int id, int quantity)
        throws OrderException{
        PriceQuote priceQuotePort = service.getPriceQuote();
        Product product = new Product();
        product.setId(1);
        product.setQuantity(4);
        try {
            float quote = priceQuotePort.getQuote(product);
            return quote;
        }catch(QuoteFault ex) {....}
    }
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

When handling web service operation input, output, and fault elements, consider converting JAXB object provided by the web service into your application objects.

Multiple Web Service references can be defined together using `@javax.xml.ws.WebServiceRefs` annotation.

Summary

In this lesson, you should have learned how to handle communications with SOAP Services, including how to:

- Describe a SOAP Web Service structure
- Create SOAP Web Services using JAX-WS API
- Create SOAP Web Service clients



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



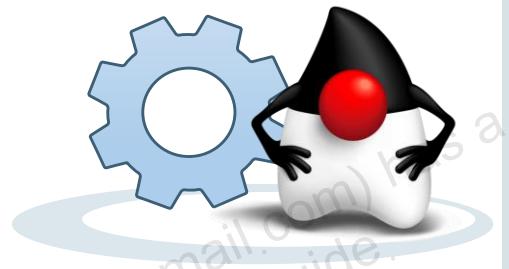
Practice Overview

This practice covers the following tasks:

- Create Price Quote Service Based on existing WSDL and XSD files
- Create Java Client Application that consumes the Price Quote Service



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Creating Java Web Applications by Using Servlets



ORACLE®



7

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this lesson, you should be able to create Java servlets and manage HTTP communications, including:

- Describe HTTP basics
- Create Java servlet classes and map them to URLs
- Handle HTTP headers, parameters, cookies
- Use servlets to handle different content types
- Manage servlet life cycle with container callback methods
- Use CDI Managed Beans
- Use WebFilters
- Implement asynchronous servlets and use NIO API
- Handle errors



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



HTTP Protocol Basics: Sending Requests

HTTP GET method:

- Transmits parameters together with the URL
- Typically is used to request initial pages where parameters are not required

```
http://www.example.com/demos/something?p1=A&p2=B
```



```
GET /demos/something?p1=A&p2=B HTTP/1.1  
Host: www.example.com
```

HTTP POST method:

- Transmits parameters as separate name-value pairs
- Is used to submit user input data

```
<form action="something" method="POST">  
<input type="text" name="p1" value="A">  
  <input type="text" name="p2" value="B">  
  <input type="submit" value="OK">  
</form>
```



```
POST /demos/something HTTP/1.1  
Host: www.example.com  
p1=A&p2=B
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The Method token indicates the method to be performed on the resource identified by the Request-URI.

Here is the list of standard HTTP Methods:

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

HTTP Protocol Basics: Getting Responses

Status code that indicates the nature of the response to the client

- 1xx codes: Informational
Example: 101 Switching Protocols
- 2xx codes: Successful Responses
Example: 200 OK
- 3xx codes – Redirection
Example: 301 Moved Permanently
- 4xx codes: Client Errors
Example: 404 Not Found
- 5xx codes: Server Errors
Example: 503 Service Unavailable



Headers that describe metadata

- Content Type
- Content Length
- Character Encoding
- Date and Time
- And so on

Optional Body of the Response

Example: Response contains 200 OK code and body will be dependent on the method used in the request.

- GET: Response body should contain requested resource content.
- HEAD: No response body is produced, just headers.
- POST: Response body should contain result of requested action.
- TRACE: Response body should echo the request message back.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

For a complete list of HTTP Protocol Status Codes and descriptions, refer to RFC-2616 standard documentation section 10: <https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>

For a complete list of HTTP Protocol Headers and descriptions refer to RFC-2616 standard documentation section 14: <https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html>

The body of an HTTP Response may not necessarily be present; it may depends upon the status of the response.

Create Servlet

Servlet class:

- Extends HttpServlet
- Is mapped to urlPattern
- Provides HTTP Request Handling operations (see next page)

```
package demos;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
@WebServlet(name = "ProductDisplay", urlPatterns = {"/products"})
public class ProductDisplayServlet extends HttpServlet {
    // Override Servlet Methods (next page)
}
```

Described with:

- Annotations or
- web.xml file or
- Both

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
           http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">
    <servlet>
        <servlet-name>ProductDisplay</servlet-name>
        <servlet-class>demos.ProductDisplayServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>ProductDisplay</servlet-name>
        <url-pattern>products</url-pattern>
    </servlet-mapping>
</web-app>
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Descriptors placed in the web.xml file take precedence over annotation-based configurations.

Servlet annotations may even be completely disabled by using the metadata-complete attribute of the web-app tag.

```
<web-app ... version="3.1" metadata-complete="true">
```

Although annotations are more convenient to use, deployment descriptors provide better flexibility, for example, providing context or initialization parameters. They could be provided via annotations, but then changes would require modifications of java sources. Therefore, it is a good use-case for applying xml descriptors.

Context Parameters are shared by all components of this Web Module.

```
<context-param>
    <param-name>cssFile</param-name>
    <param-value>some.css</param-value>
</context-param>
```

Initialization Parameters are Servlet specific.

```
<servlet>
    <servlet-name>ProductDisplay</servlet-name>
    <servlet-class>demos.Product</servlet-class>
    <init-param>
        <param-name>defaultLanguage</param-name>
        <param-value>en</param-value>
    </init-param>
</servlet>
```

Servlet may access such parameters:

```
package demos;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
@WebServlet(name = "ProductDisplay", urlPatterns = {"/products"})
public class ProductDisplayServlet extends HttpServlet {
    @Inject
    private ServletContext context;
    protected void processRequest(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {
        String p1 = context.getInitParameter("defaultLanguage");
        String p2 = context.getInitParameter("cssFile");
    }
}
```

Override Servlet Request Handling Operations

Override `service`

- No distinction between HTTP Methods is required.
- Disable all other Request Handling methods.

Override `doGet` `doPost` `doPut` `doDelete` and so on

- Handle specific HTTP Methods differently
- Can be used together with `processRequest` method

Call single operation from all `doXXXX` methods: `processRequest`

- Use common as well as specific Request Handling methods

These operations use identical parameters and throw identical exceptions.

- `HttpServletRequest`
- `HttpServletResponse`
- `ServletException`
- `IOException`

```
@WebServlet(name = "ProductDisplay", urlPatterns = {"/products"})
public class ProductDisplayServlet extends HttpServlet {
    protected void service(HttpServletRequest request,
                           HttpServletResponse response)
        throws ServletException, IOException { }

    protected void processRequest ...
    protected void doGet ...
    protected void doPost ...
}
```

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Provide Request Handling Logic

Servlet request handling operations

- Use `HttpServletRequest` object to find information about the call your servlet has received.
- Use `HttpServletResponse` object to produce output for your servlet invoker.

```
...
protected void processRequest(HttpServletRequest request,
                             HttpServletResponse response)
                             throws ServletException, IOException {
    String httpMethod = request.getMethod();
    response.setContentType("text/html; charset=UTF-8");
    PrintWriter out = response.getWriter();
    out.println("<html><body>Servlet Received "+httpMethod+" call</body></html>");
    out.close();
}
...
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Retrieve Request Headers

HTTP Request headers contain information about invoking client.

- Use the `getHeaderNames` method to retrieve all request header names.
- Iterate through enumeration of header names and get their values.

```
...
protected void processRequest(HttpServletRequest request,
                             HttpServletResponse response)
                             throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    PrintWriter out = response.getWriter();
    out.println("<html><body>");
    Enumeration<String> headers = request.getHeaderNames();
    while (headers.hasMoreElements()) {
        String name = headers.nextElement();
        String value = request.getHeader(name);
        out.println("Header:"+name+"="+value+"<br>");
    }
    out.close();
}
...
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

It is important to be able to handle HTTP Protocol Headers in your Servlet code, because they contain valuable information about the invoking client. You can use headers to find what browser, operation system, country or language and so on that the client uses.

Retrieve Parameters

Servlet can get parameters submitted by the client.

- Use the `getParameterNames` method to retrieve all request parameter names.
- Iterate through enumeration of parameter names and get their values.

```
...
protected void processRequest(HttpServletRequest request,
                             HttpServletResponse response)
                             throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    PrintWriter out = response.getWriter();
    out.println("<html><body>");
    Enumeration<String> parameters = request.getParameterNames();
    while (parameters.hasMoreElements()) {
        String name = parameters.nextElement();
        String value = request.getParameter(name);
        out.println("Parameter:"+name+"="+value+"<br>");
    }
    out.close();
}
...
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Parameters may be submitted to your servlet from the client in different ways, such as simple GET Method URLs, or POST method forms and so on.

The `HttpServletRequest` object would present you this parameters in the same fashion, regardless of the specific HTTP method that was used in this request.

Presence of the parameter does not necessarily imply that it actually has nonempty or valid value. Therefore, before using parameter values it is considered important to validate them first.

HTTP Protocol transmits all parameters as Strings, so you may have to parse values and convert data types.

```
String value = request.getParameter(name);
if (value != null && value.length() != 0) {
    int x = Integer.parseInt(value);
}
```

Use Cookies

Cookies can store state information in the client tier.

- Use the `getCookies` method to retrieve all cookies.
- Create a new `Cookie` Object.
- Set `Cookie` expire time to 1 hour.

```
...
protected void processRequest(HttpServletRequest request,
                             HttpServletResponse response)
                             throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    PrintWriter out = response.getWriter();
    out.println("<html><body>");
    Cookie[] cookies = request.getCookies();
    if(cookies != null) {
        for(Cookie cookie : cookies) {
            String name = cookie.getName();
            String value = cookie.getValue();
            out.println("Parameter:"+name+"="+value+"<br>");
        }
        Cookie cookie = new Cookie("aName","aValue");
        cookie.setMaxAge(60*60); //Expire in an hour
        response.addCookie(cookie);
    }
    out.close();
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Cookies are used to store state information within the client tier. They are known not to be secure, have scalability limitations, and may be entirely disabled.

Alternatively, "smarter" JavaScript client can use new HTML5 standard called local storage. However, data stored in the HTML5 local storage cannot be directly accessed by Java EE Web Container, because it is handled exclusively by JavaScript within the client tier.

Cookies can be set directly by the Java EE Web Container component such as Servlets, JSPs, JSFs, and so on. When creating cookies, consider setting their expiry time. If you need to remove a cookie, just set its expiry time to any point in the past.

Produce Different Content Types

Output produced by the Servlet does not have to be HTML or text.

- Set HTTP Headers to inform invoker about the nature of content your servlet is going to generate.
 - Content-Type
 - Content-Length (optional, but could be a good idea)
- Use PrintWriter or ServletOutputStream objects.

This example is artificial.

- A static file can be dowloaded without the use of servlet.
- Consider generating this image dynamically.

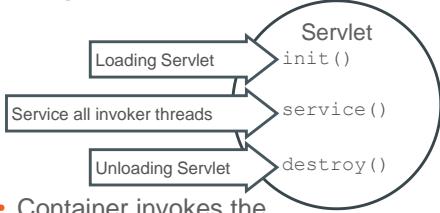
```
...
protected void processRequest(HttpServletRequest request,
                               HttpServletResponse response)
                               throws ServletException, IOException {
    response.setContentType("image/jpeg");
    File f = new File("somePicture.jpeg");
    response.setContentLength((int)f.length());
    ServletOutputStream out = response.getOutputStream();
    Files.copy(f.toPath(), out);
    out.close();
}
...
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Manage Servlet Life Cycle with Container Callbacks

A single servlet instance is shared between all callers.



- Container invokes the `init(ServletConfig)` method to notify that servlet is broad into service.
- `init()` is convenience method that can be overridden so that there's no need to call `super.init(config)`.
- Container invokes `destroy` method to notify that servlet is been taken out of service.

```

package demos.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
@WebServlet(urlPatterns = {"/*products"})
public class ProductDisplayServlet extends HttpServlet {
    @Override
    public void init() {...}
    // or
    public void init(ServletConfig config) {
        super.init(config);
        ...
    }
    @Override
    public void destroy() {...}

    protected void processRequest ...
}

```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

A single instance of the servlet is initialized by WebContainer and at this stage container invokes `init` method upon the servlet. Overriding this method, allows you to run custom code at the initialization phase of the servlet life cycle.

Each call the servlet receives is served by the same instance of the servlet class. Web Container uses different threads (one thread per call) to trigger Request Processing of this servlet.

When container offloads servlet (for example, at shutdown) it invokes `destroy` method upon your servlet. Overriding this method allows you to run custom code at the destruction phase of the servlet life cycle.

CDI Beans

- Define CDI Beans.
- Control CDI processing.

```
package demos;
```

```
@SessionScoped
```

```
public class MyBean implements Serializable {
    public void doSomething() {...}
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                           http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
       version="1.1" bean-discovery-mode="annotated">
</beans>
```

Inject your beans.

Just use them as required.

```
package demos;
...
@WebServlet(urlPatterns = {" /products"})
public class ProductDisplayServlet extends HttpServlet {
    @Inject
    private MyBean mb;
    protected void processRequest(HttpServletRequest request,
                                  HttpServletResponse response)
        throws ServletException, IOException {
        mb.doSomething();
    }
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

A `beans.xml` file is a deployment descriptor, which can be used to describe CDI beans, but may simply rely on annotations.

The `bean-discovery-mode` attribute controls how CDI beans are processed. The recommended setting is "annotated", that allows all annotated beans to be automatically used.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                           http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
       version="1.1" bean-discovery-mode="all">
    ...
</beans>
```

CDI can manage and inject any bean in an explicit archive, except those annotated with `@Vetoed`.

An implicit bean archive is an archive that contains some beans annotated with a scope type, contains no `beans.xml` deployment descriptor, or contains a `beans.xml` deployment descriptor with the `bean-discovery-mode` attribute set to `annotated`.

In an implicit archive, CDI can only manage and inject beans annotated with a scope type.

For a web application, the `beans.xml` deployment descriptor, if present, must be in the `WEB-INF` directory. For EJB modules or JAR files, the `beans.xml` deployment descriptor, if present, must be in the `META-INF` directory.

HTTP Session Tracking

To join requests into the same session object, Web Container has to identify that these requests are arriving from the same client using the `jsessionid` cookie or the `url` parameter.

- Cookies are not reliable.
- The `url` parameter is preferred.

Use the `encodeURL` method to append the `jsessionid` parameter to `url`.

```
protected void processRequest(HttpServletRequest request,
                             HttpServletResponse response)
                             throws ServletException, IOException {
    String url = response.encodeURL("displayProduct");
    PrintWriter out = response.getWriter();
    out.println("<html><body>");
    out.println("<A HREF='"+url+"'>Show Product</a>");
    out.println("</body></html>");
    out.close();
}
```

This will generate an HTML page that can pass `jsessionid` back to the server with the next request.

```
<html><body>
<A HREF='displayProduct;jsessionid=147JQ'>Show Product</a>
</body></html>
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Although there is no problem for a Web Container to identify Request or Application Scopes. Session scope does present an issue: how to identify if request are coming from the same client and, therefore, should be related to specific session instance. Therefore, for each request, the server must be able to identify the specific browser to select the correct session object. This session binding is performed using cookies or URL rewriting. Cookies known not to be particularly reliable, so URL rewriting is a preferred method. For example, if the web component generates HTML such as the following fragment, the "displayProduct" URI must be rewritten to expose the session ID:

`Show Product`

The following code shows how the "displayProduct" URI would be rewritten:

`Show Product`

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

Web Container Life Cycle Events

A Web Listener can implement any number of life cycle event handling interfaces.

```
@WebListener
public class SomeListener implements
HttpSessionListener, ServletContextListener {
    public void sessionCreated(HttpSessionEvent e) {...}
    public void sessionDestroyed(HttpSessionEvent e) {...}
    public void contextInitialized(ServletContextEvent e) {...}
    public void contextDestroyed(ServletContextEvent e) {...}
}
```

Objects stored in a session can receive event notifications:

```
@SessionScoped
public class SomeObject implements Serializable,
HttpSessionActivationListener, HttpSessionBindingListener {
    public void sessionDidActivate(HttpSessionEvent e) {...}
    public void sessionWillPassivate(HttpSessionEvent e) {...}
    public void valueBound(HttpSessionBindingEvent e) {...}
    public void valueUnbound(HttpSessionBindingEvent e) {...}
}
```

ServletRequestListener
ServletRequestAttributeListener

Request

HttpSessionListener
HttpSessionActivationListener
HttpSessionIdListener
HttpSessionBindingListener
HttpSessionAttributeListener

HTTP Session

ServletContextListener
ServletContextAttributeListener

Application



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Request Context Listeners

The ServletRequestListener interface is designed for receiving notification events about requests coming into and going out of scope of a web application.

A ServletRequest is defined as coming into scope of a web application when it is about to enter the first servlet or filter of the web application, and as going out of scope as it exits the last servlet or the first filter in the chain.

Implementations of this interface are invoked at their `requestInitialized(javax.servlet.ServletRequestEvent)` method in the order in which they have been declared, and at their `requestDestroyed(javax.servlet.ServletRequestEvent)` method in reverse order.

The ServletRequestAttributeListener interface is designed for receiving notification events about the ServletRequest attribute changes.

Notifications will be generated while the request is within the scope of the web application. A ServletRequest is defined as coming into scope of a web application when it is about to enter the first servlet or filter of the web application, and as going out of scope when it exits the last servlet or the first filter in the chain.

This interface requires implementor to override following methods:

`attributeAdded(ServletRequestAttributeEvent event)`: To receive notification that an attribute has been added to the ServletRequest

`attributeRemoved(ServletRequestAttributeEvent event)`: To receive notification that an attribute has been removed from the ServletRequest

`attributeReplaced(ServletRequestAttributeEvent event)`: To receive notification that an attribute has been updated in the ServletRequest

Http Session Context Listeners

`HttpSessionListener` interface is designed for receiving notification events about HttpSession life cycle changes.

Implementations of this interface are invoked at their

`sessionCreated(javax.servlet.http.HttpSessionEvent)` method in the order in which they have been declared, and at their

`sessionDestroyed(javax.servlet.http.HttpSessionEvent)` method in reverse order.

The `HttpSessionIdListener` interface is designed for receiving notification events about HttpSession ID changes.

This interface requires implementor to override the

`sessionIdChanged(HttpSessionEvent event, String oldSessionId)` method to receives notification that session ID has been changed in a session.

`HttpSessionActivationListener` is designed to notify objects that are bound to the session that container will be passivating or activating the session. A container that migrates session between VMs or persists sessions is required to notify all attributes bound to sessions implementing `HttpSessionActivationListener`.

This interface requires implementor to override following methods:

`sessionDidActivate(HttpSessionEvent se)`: To receive notification that the session has just been activated

`sessionWillPassivate(HttpSessionEvent se)`: To receive notification that the session is about to be passivated

`HttpSessionBindingListener` causes an object to be notified when it is bound to or unbound from a session. The object is notified by an `HttpSessionBindingEvent` object. This may be as a result of a servlet programmer explicitly unbinding an attribute from a session, due to a session being invalidated, or due to a session timing out.

This interface requires implementor to override following methods:

`valueBound(HttpSessionBindingEvent event)`: Notifies the object that it is being bound to a session and identifies the session

`valueUnbound(HttpSessionBindingEvent event)`: Notifies the object that it is being unbound from a session and identifies the session

The HttpSessionAttributeListener interface is designed for receiving notification events about the HttpSession attribute changes.

This interface requires implementor to override following methods:

attributeAdded(HttpSessionBindingEvent event): To receive notification that an attribute has been added to the session

attributeRemoved(HttpSessionBindingEvent event): To receive notification that an attribute has been removed from the session

attributeReplaced(HttpSessionBindingEvent event): To receive notification that an attribute has been updated in the session

Web Application Context Listeners

The ServletContextListener interface is designed for receiving notification events about ServletContext life cycle changes.

Implementations of this interface are invoked at their

contextInitialized (javax.servlet.ServletContextEvent) method in the order in which they have been declared, and at their

contextDestroyed (javax.servlet.ServletContextEvent) method in reverse order.

The ServletContextAttributeListener interface is designed for receiving notification events about the ServletContext attribute changes.

This interface requires implementor to override following methods:

attributeAdded (ServletContextAttributeEvent event): To receive notification that an attribute has been added to the ServletContext

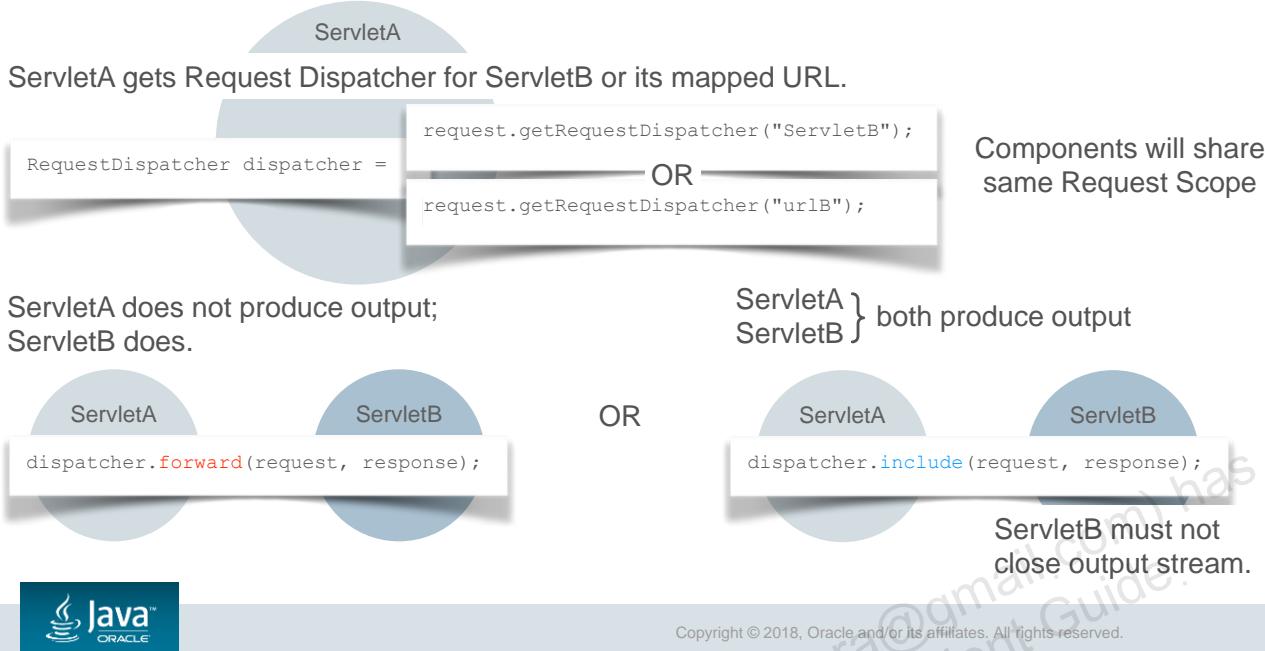
attributeRemoved (ServletContextAttributeEvent event): To receive notification that an attribute has been removed from the ServletContext

attributeReplaced (ServletContextAttributeEvent event): To receive notification that an attribute has been updated in the ServletContext

All classes implementing listeners (except HttpSessionActivationListener and SessionBindingListener) must be either declared in the deployment descriptor of the web application, annotated with WebListener, or registered via one of the addListener methods defined on ServletContext.

HttpSessionActivationListener is simply implemented by object added to the session, so does not require additional registration.

Request Dispatcher



Request Dispatchers is a controller for a Servlet, JSP page or any other Java resource mapped to URL in the Web Container.

If a servlet has no URI because it was not configured to have a url-pattern, then an instance of a `RequestDispatcher` object can be retrieved from the runtime context of the servlet by giving the name of the target servlet.

According to the servlet specification, a URI that is passed as an argument to `ServletContext.getRequestDispatcher` must begin with a slash (/). However, 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 shortcut for the context root. The presence of the slash is what prevents the need to give the context root itself.

The `ServletRequest.getRequestDispatcher` method can take a relative URL as long as it is within the application context.

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.

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.

Note

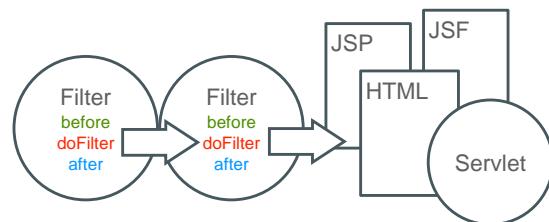
The forward method cannot be used if the servlet has already begun to produce output. This means that after the servlet has called the `response.getWriter` method, it is too late to use the forward method, regardless of whether the servlet has used the writer itself. This is because the web container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

Included components should not call the close method.

Servlet Filters

Web Filters allow you to apply reusable logic to a number of different web components, in a way that is transparent to both client and components themselves.

- Client invokes Web Component as usual.
- Filter could be mapped to specific Web Component or URL pattern.
- More than one Filter can be mapped to same URL.
- Filters intercept calls on URLs they were mapped to.
- Filter can perform actions:
 - Before passing control to the next component in the chain via doFilter
 - After the doFilter method returns
- Request Scope is shared between all components in the filter chain.



```

@WebFilter(filterName = "SomeFilter", urlPatterns = {"/*"})
public class SomeFilter implements Filter {
    public void doFilter(ServletRequest request,
                        ServletResponse response,
                        FilterChain chain)
        throws IOException, ServletException {
        // Perform "Before" Actions
        try {
            chain.doFilter(request, response);
        } catch (Throwable t) {
            // Perform "After Error" Actions
        }
        // Perform "After" Actions
    }
}
  
```

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



A filter is an object that can transform the header and content (or both) of a request or response. Filters differ from web components in that filters usually do not themselves create a response. Instead, a filter provides functionality that can be "attached" to any kind of web resource. Consequently, a filter should not have any dependencies on a web resource for which it is acting as a filter; this way, it can be composed with more than one type of web resource.

The main tasks that a filter can perform are as follows.

- Query the request and act accordingly.
- Block the request-and-response pair from passing any further.
- Modify the request headers and data. You do this by providing a customized version of the request.
- Modify the response headers and data. You do this by providing a customized version of the response.
- Interact with external resources.

Applications of filters include authentication, logging, image conversion, data compression, encryption, tokenizing streams, XML transformations, and so on.

You can configure a web resource to be filtered by a chain of zero, one, or more filters in a specific order. This chain is specified when the web application containing the component is deployed and is instantiated when a web container loads the component.

Filter can be applied to handle only specific types of calls using DispatcherType attribute. Valid Dispatcher types are:

- REQUEST (default) – filter is applied to all requests coming from clients
- ASYNC – filter is applied only to asynchronous requests
- ERROR – filter is applied only to those requests that are processed with the error page mechanism
- FORWARD – filter is applied only to requests that are forwarded to other components
- INCLUDE – filter is applied only to requests that include other components

You can only define the order of filters in the chain using web.xml file.

Asynchronous Servlets

Use the `AsyncContext` object to handle "slow" resources, in a separate thread.

- Enable Asynchronous Support.
- Start Asynchronous Context.
- Operate on Asynchronous Context using:
 - start
 - getRequest
 - getResponse
 - complete
 - dispatch

```
@WebServlet(urlPatterns = {"/*"}, asyncSupported=true)
public class AsynchronousServlet extends HttpServlet {
    protected void processRequest(HttpServletRequest request,
                                  HttpServletResponse response)
        throws ServletException, IOException {
        AsyncContext ac = request.startAsync();
        ac.start(new Runnable() { public void run() {
            HttpServletRequest request = ac.getRequest();
            HttpServletResponse response = ac.getResponse();
            // perform request processing and produce response
            ac.complete();
        }
    }
}
```

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Java EE provides asynchronous processing support for servlets and filters. If a servlet or a filter reaches a potentially blocking operation when processing a request, it can assign the operation to an asynchronous execution context and return the thread associated with the request immediately to the container without generating a response. The blocking operation completes in the asynchronous execution context in a different thread, which can generate a response or dispatch the request to another servlet.

The `AsyncContext` class defines the following operations:

`void start(Runnable run)` : Container provides a different thread in which the blocking operation can be processed.

You provide code for the blocking operation as a class that implements the `Runnable` interface. You can provide this class as an inner class when calling the `start` method or use another mechanism to pass the `AsyncContext` instance to your class.

`ServletRequest getRequest()` : Returns the request used to initialize this asynchronous context. In the example above, the request is the same as in the service method.

You can use this method inside the asynchronous context to obtain parameters from the request.

`ServletResponse getResponse()` : Returns the response used to initialize this asynchronous context. In the example above, the response is the same as in the service method.

You can use this method inside the asynchronous context to write to the response with the results of the blocking operation.

`void complete():` Completes the asynchronous operation and closes the response associated with this asynchronous context

You call this method after writing to the response object inside the asynchronous context.

`void dispatch(String path):` Dispatches the request and response objects to the given path.

You use this method to have another servlet write to the response after the blocking operation completes.

Nonblocking I/O

Use NIO to handle "slow" network clients without blocking Web Container.

- Enable Asynchronous Servlet Processing.
- Use ServletInputStream to setReadListener that provides:
 - onDataAvailable
 - onAllDataRead
 - onError
- Use ServletOutputStream to setWriteListener that provides:
 - onWrite
 - onError

```
@WebServlet(urlPatterns = {"/aServlet"}, asyncSupported=true)
public class AsynchronousServlet extends HttpServlet {
    protected void processRequest(HttpServletRequest request,
                                  HttpServletResponse response)
                                  throws ServletException, IOException {
        AsyncContext ac = request.startAsync();
        ServletInputStream in = request.getInputStream();
        in.setReadListener(new ReadListener() {
            @Override
            public void onDataAvailable() { /*read request data*/ }
            @Override
            public void onAllDataRead() {
                /* process data when it becomes available */
                ac.complete();
            }
            @Override
            public void onError(Throwable t) { /*handle errors*/ }
        });
    }
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Web containers in application servers normally use a server thread per client request. To develop scalable web applications, you must ensure that threads associated with client requests are never sitting idle waiting for a blocking operation to complete. Asynchronous Processing provides a mechanism to execute application-specific blocking operations in a new thread, returning the thread associated with the request immediately to the container. Even if you use asynchronous processing for all the application-specific blocking operations inside your service methods, threads associated with client requests can be momentarily sitting idle because of input/output considerations.

For example, if a client is submitting a large HTTP POST request over a slow network connection, the server can read the request faster than the client can provide it. Using traditional I/O, the container thread associated with this request would be sometimes sitting idle waiting for the rest of the request.

Java EE provides nonblocking I/O support for servlets and filters when processing requests in asynchronous mode.

The following steps summarize how to use nonblocking I/O to process requests and write responses inside service methods.

- Put the request in asynchronous mode as described in Asynchronous Processing.
- Obtain an input stream and/or an output stream from the request and response objects in the service method.
- Assign a read listener to the input stream and/or a write listener to the output stream.
- Process the request and the response inside the listener's callback methods.

ServletInputStream supports following NIO operations:

`void setReadListener(ReadListener rl)` : Associates this input stream with a listener object that contains callback methods to read data asynchronously. You provide the listener object as an anonymous class or use another mechanism to pass the input stream to the read listener object.

`boolean isReady()` : Returns true if data can be read without blocking

`boolean isFinished()` : Returns true when all the data has been read

javax.servlet.ServletOutputStream supports following NIO operations:

`void setWriteListener(WriteListener wl)` : Associates this output stream with a listener object that contains callback methods to write data asynchronously. You provide the write listener object as an anonymous class or use another mechanism to pass the output stream to the write listener object.

`boolean isReady()` : Returns true if data can be written without blocking

Listener Interfaces for Nonblocking I/O Support are ReadListener and WriteListener.

ReadListener supports the following NIO operations:

```
void onDataAvailable()  
void onAllDataRead()  
void onError(Throwable t)
```

A ServletInputStream instance calls these methods on its listener when there is data available to read, when all the data has been read, or when there is an error.

WriteListener supports following NIO operations:

```
void onWritePossible()  
void onError(Throwable t)
```

A ServletOutputStream instance calls these methods on its listener when it is possible to write data without blocking or when there is an error.

Example:

```
@WebServlet(urlPatterns = {"/aServlet"}, asyncSupported=true)
public class AsynchronousServlet extends HttpServlet {
    protected void processRequest(HttpServletRequest request,
                                  HttpServletResponse response)
        throws ServletException, IOException {
        AsyncContext ac = requset.startAsync();
        ServletInputStream in = request.getInputStream();
        in.setReadListener(new ReadListener() {
            byte buffer[] = new byte[4*1024];
            StringBuilder sb = new StringBuilder();
            @Override
            public void onDataAvailable() {
                try {
                    do {
                        int length = input.read(buffer);
                        sb.append(new String(buffer, 0, length));
                    } while(in.isReady());
                } catch (IOException ex) { ... }
            }
            @Override
            public void onAllDataRead() {
                try {
                    ac.getResponse().getWriter().write("...the response...");
                } catch (IOException ex) { ... }
                ac.complete();
            }
            @Override
            public void onError(Throwable t) { ... } });
    }
}
```

Handle Errors

Servlets may produce errors by either:

- Sending HTTP error code, or
- Throwing an exception

`web.xml` describes error handles for:

- Specific HTTP response codes or
- Java Exceptions

Any web component can be designated as an error handler.

An error-handling servlet can access information about an error by using request attributes.



```
response.sendError(404,"This page is not here");
throw new ServletException("Big Issue");
```

```
<error-page>
    <error-code>404</error-code>
    <location>/ErrorPage.html</location>
</error-page>
<error-page>
    <exception-type>java.lang.Throwable</exception-type>
    <location>/ErrorServlet</location>
</error-page>
```

```
request.getAttribute("javax.servlet.error.status_code");
request.getAttribute("javax.servlet.error.exception_type");
request.getAttribute("javax.servlet.error.message");
request.getAttribute("javax.servlet.error.request_uri");
request.getAttribute("javax.servlet.error.exception");
request.getAttribute("javax.servlet.error.servlet_name");
```

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The following is the list of request attributes that an error-handling servlet can access to analyze the nature of error/exception.

`javax.servlet.error.status_code`: This attribute gives a status code, which can be stored and analyzed after storing in a `java.lang.Integer` data type.

`javax.servlet.error.exception_type`: This attribute gives information about the exception type, which can be stored and analyzed after storing in a `java.lang.Class` data type.

`javax.servlet.error.message`: This attribute gives information about the exact error message, which can be stored and analyzed after storing in a `java.lang.String` data type.

`javax.servlet.error.request_uri`: This attribute gives information about the URL calling the servlet and it can be stored and analyzed after storing in a `java.lang.String` data type.

`javax.servlet.error.exception`: This attribute gives information about an exception raised, which can be stored and analyzed after storing in a `java.lang.Throwable` data type.

`javax.servlet.error.servlet_name`: This attribute gives a servlet name, which can be stored and analyzed after storing in a `java.lang.String` data type.

Summary

In this lesson, you should have learned how to create Java Servlets and manage HTTP communications, including:

- Describe HTTP basics
- Create Java servlet classes and map them to URLs
- Handle HTTP headers, parameters, cookies
- Use servlets to handle different content types
- Manage servlet life cycle with container callback methods
- Use CDI Managed Beans
- Use WebFilters
- Implement asynchronous servlets and use NIO API
- Handle errors



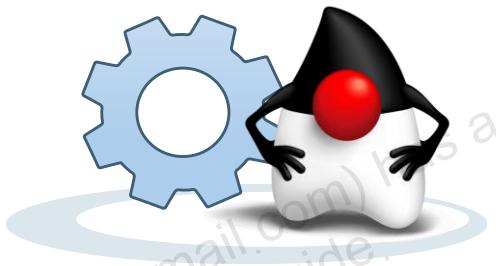
Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Practice Overview

This practice covers the following tasks:

- Create Web Application that can search and display products
- Create HTML Product Search Form that submits information to Java Servlet
- Create Java Servlet to execute search and display list of products



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Creating Java Web Applications by Using JSPs

8



ORACLE®

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Objectives

After completing this lesson, you should be able to create Java Server Pages. You should be able to:

- Describe JSP life cycle
- Describe JSP syntax
- Use Expression Language (EL)
- Use CDI Beans
- Use Tag Libraries
- Handle errors



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

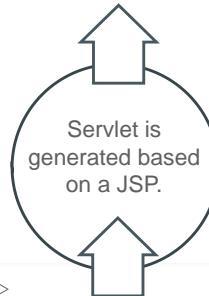


Create Java Server Page

JSP Page is designed as a page, but it runs as a servlet.

- Created as either JSP or XML document
- Has same runtime capabilities as a Servlet
- No Servlet URL mapping required; invokers use the URL of a JSP page, but a generated servlet is invoked instead
- Allows web UI developers to focus on presentation
- All servlet coding techniques applicable:
 - CDI Beans
 - Web Filters
 - Container callback operations
 - Life-cycle event listeners

Servlet is running in a JEE Web Container.



Developer designs JSP document.

```
<%-- Comments --%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html><body>
    <%String p1 = request.getParameter("p1");%>
    Hello <%=p1%>
</body></html>
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

JavaServer Pages (JSP) technology enables you to insert snippets of servlet code directly into a text-based document. A JSP page is a text-based document that contains two types of text:

- Static data, which can be expressed in any text-based format, such as HTML or XML
- JSP elements, which determine how the page constructs dynamic content

Java Server Page Syntax

JSP elements:

- Any text, HTML, XML, and so on
- Comments <%-- --%>
- Directives
 - <%@page %>
 - <%@include %>
 - <%@taglib %>
- Declarations <%! %>
- Scriptlets <% %>
- Expressions <%= %>

```

<%-- Comments --%>
<%@page import="demos.SomeBean"%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%! private SomeBean some; %>
<!DOCTYPE html>
<html><body>
  <% String p1 = request.getParameter("p1");
     if (p1 != null & p1.length() != 0) {
       some.setSomething(p1);
     }%
  Hello <%=some.getSomething()%>
  <%}else{%
    Hello World!
  <%}%>
</body></html>

```

Scriptlets and Expressions may reference standard Servlet objects:

- request, response, out, session, application, config, pageContext, page, exception



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Directives

- <%@page %>: Defines global page-dependent attributes, such as session tracking, error page, buffering requirements, content-type headers, java imports and so on
- <%@include %>: Includes a static file
- <%@taglib %>: References tag library

Declarations

<%! %>: Contain any java code that a developer wants to include outside the service method of a future servlet that is going to be created out of this Java Server Page. This could be other methods, or variable declarations.

Scriptlets

<% %>: Contain any java code that developer wants to place into the service method of a future servlet

Expressions

<%= %>: Outputting Java variables. Similar to `out.println()`. For example, Expression `<%=p1%>` gives you the same output as Scriptlet `<% out.println(p1); %>`

Scriptlets and Expressions may reference standard Servlet objects:

- **request:** This is the HttpServletRequest object associated with the request.
- **response:** This is the HttpServletResponse object associated with the response to the client.
- **out:** This is the PrintWriter object that is used to send output to the client.
- **session:** This is the HttpSession object associated with the request.
- **application:** This is the ServletContext object associated with application context.
- **config:** This is the ServletConfig object associated with the page.
- **pageContext:** This encapsulates use of server-specific features such as high performance JspWriters.
- **page:** This is simply a synonym for `this`, and is used to call the methods defined by the translated servlet class.
- **exception:** The Exception object allows the exception data to be accessed by a designated JSP.

Java Server Page XML Syntax

JSP XML Elements

- <jsp:element></jsp:element>
- Comments <!-- -->
- Directives
 - <jsp:directive.page/>
 - <jsp:directive.include/>
 - <jsp:directive.taglib/>
- Declarations <jsp:declaration></jsp:declaration>
- Scriptlets <jsp:scriptlet></jsp:scriptlet>
- Expressions <jsp:expression></jsp:expression>

Functionally choice of syntax makes no difference; either way pages get same abilities.

JSP XML Document root element must be **jsp:root**

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Comments -->
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page">
  <jsp:directive.page import="demos.SomeBean"/>
  <jsp:directive.page contentType="text/html"
    pageEncoding="UTF-8"/>
  <jsp:declaration>
    private SomeBean some;
  </jsp:declaration>
  <jsp:scriptlet>
    String p1 = request.getParameter("p1");
  </jsp:scriptlet>
  <jsp:element name="text">
    <jsp:attribute name="lang">EN</jsp:attribute>
    <jsp:body>Hello</jsp:body>
  </jsp:element>
  <jsp:expression>p1</jsp:expression>
</jsp:root>
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Expression Language

- Originally, Expression Language (EL) emerged as a part of JSP specification.
- EL 3.0 is a standard of its own. It can be used with frameworks such as JSP, JSF, CDI and so on.

- Two EL syntaxes:

- Immediate (JSP)

`${expression}`

(used in this lesson)

- Deferred (JSF)

`#{expression}`

(used in the Java Server Faces lesson)

- Examples of expressions (can use " or ' quotation marks) :

- Access bean properties

`${someBean.someProperty}`

`${someBean['someProperty']}`

- Invoke operations

`${someBean.someMethod('parameter')}`

- Handle arrays

`${someArray[index]}`

- Evaluate Lambdas

`${(x,y) -> x+y}`



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Expression Language 3.0 standard is JSR-341:

<https://jcp.org/en/jsr/detail?id=341>

Immediate Expressions:

- Are mostly used by JSP pages
- Could also be used by JSF templates
- (from a JSF perspective) Are evaluated at the beginning of the life cycle of a JSF page
- (in JSF applications) Can only provide read-only access to bean values

Deferred Expressions are:

- In sync with the JSF life cycle. This means that an EL expression in deferred EL is evaluated at different points in the JSF life cycle.
- In JSF, applications provide full (read and write) access to bean values.

Web application configuration file (web.xml) or a specific page (via page directive) can turn scripting evaluation OFF. By default it is ON.

```
<jsp-property-group>
    <url-pattern>*.jsp</url-pattern>
    <scripting-invalid>true</scripting-invalid>
</jsp-property-group>
```

Expression Language Operators

| | |
|-----------------------|---|
| . | Access property, method or Map entry |
| , | Statement separator |
| [] | Access an array or List element |
| () | Method parameters or change of precedence |
| + - * div % mod | Arithmetic operators |
| = | Assignment operator |
| == != < > <= >= | |
| eq ne lt gt le ge | Relational operators |
| && ! and or not | Logical operators |
| ?: | Conditional operator |
| empty | Test for empty variable values |
| -> | Lambda Token |



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Expression Language 3.0 standard is JSR-341:

<https://jcp.org/en/jsr/detail?id=341>

The following words are reserved for the language and must not be used as identifiers.

and eq gt true instanceof or ne le false empty not lt ge null div mod

Note that some of these words are not in the language now, but they may be in the future, so developers must avoid using these words.

JSP Scopes and Implicit Objects

JSP expressions allow you to reference:

- Memory scopes

| | |
|-------------------------------|---|
| <code>requestScope</code> | All pages processing the same request. |
| <code>pageScope</code> | Similar to <code>requestScope</code> , but limited to just one page, if request is forwarded. |
| <code>sessionScope</code> | All pages processing the all requests of the same session. |
| <code>applicationScope</code> | All pages in the same application. |

- Implicit Objects

| | |
|--------------------------|---|
| <code>out</code> | <code>javax.servlet.jsp.JspWriter</code> |
| <code>request</code> | <code>javax.servlet.http.HttpServletRequest</code> |
| <code>response</code> | <code>javax.servlet.http.HttpServletResponse</code> |
| <code>session</code> | <code>javax.servlet.http.HttpSession</code> |
| <code>application</code> | <code>javax.servlet.ServletContext</code> |
| <code>exception</code> | <code>javax.servlet.jsp.JspException</code> |
| <code>page</code> | this (current servlet instance) |
| <code>pageContext</code> | <code>javax.servlet.jsp.PageContext</code> |
| <code>config</code> | <code>javax.servlet.ServletConfig</code> |

| | |
|---------------------------|---|
| <code>initParam</code> | Initialization parameter as String |
| <code>param</code> | Access parameters as String objects |
| <code>paramValues</code> | Access parameters as Collection of String objects |
| <code>header</code> | Access HTTP Headers as String objects |
| <code>headerValues</code> | Access HTTP Headers as Collection of String objects |
| <code>cookie</code> | Access cookies as Strings |

- Examples

```
 ${param.p_name}
 ${requestScope.someBean.someProperty}
 ${exception.message}
```

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



You can access other object via the `pageContext` object:

| | |
|--------------------------|---|
| <code>out</code> | <code>pageContext.out</code> |
| <code>Request</code> | <code>pageContext.request</code> |
| <code>Response</code> | <code>pageContext.response</code> |
| <code>Session</code> | <code>pageContext.session</code> |
| <code>application</code> | <code>pageContext.servletContext</code> |

Often expressions are interchangeable and allow you to access same object in different ways:

```
 ${exception.message}      ${requestScope['javax.servlet.error.message']}
    ${request['javax.servlet.error.message']}
```

Use CDI Beans in JSPs

Declare CDI Bean with @Named annotation.

```
package demos;  
@Named("sb")  
@RequestScoped  
public class SomeBean {  
    private String something;  
    public void setSomething(String value) {  
        something = value;  
    }  
    public String getSomething() {  
        return something;  
    }  
}
```

Just use the bean via EL code.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>  
<!DOCTYPE html>  
<html><body>  
    ${sb.something} = param.p1  
    Something is: ${sb.something}  
</body></html>
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Before CDI became available, JSP pages had to employ the `<jsp:useBean>` tag or write script code to access Java Beans.

CDI make this process much easier.

Here is an example that is the equivalent of the code example on the slide above, using the `jsp:useBean` `jsp:getProperty` and `jsp:setProperty` elements, without using EL 3.0 expressions:

```
<jsp:useBean id="sb" scope="request" class="demos.SomeBean"/>  
<jsp:setProperty name="sb" property="something"  
value='<%=request.getParameter("p1") %>' />  
<jsp:getProperty name="sb" property="something"/>
```

Standard Tag Library (JSTL)

Java Server Pages allow the developer to mix Java code with page markups.

- However, a page appears to be better structured when direct Java code is replaced with tags.
- Behind each tag, there is a Handler Java class with predefined functionality described by the Tag Library.
- Use the taglib page directive to reference libraries.

JSTL Core Library Example

- **Include JSTL Core library.**
- Use tags:
 - **Iterate through collection**
 - **Output values**
 - **Test conditions**
 - And so on

```
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<c:forEach var="product" items="${order.productList}">
  <c:out value="${product.name}" /><br>
  <c:if test="${product.discount > 0}">Sale</c:if>
  <c:out value="${product.price}" /><br>
</c:forEach>
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

A tag library is a collection of actions that encapsulates functionality to be used from within a JSP page.

JSTL is known as a standard tag library that covers basic functional requirements for processing page logic, formatting values, manipulating XML, and accessing database via SQL statements.

Java EE 7 provides several prewritten custom tags known as the JavaServer Pages Standard Tag Library (JSTL).

These libraries are grouped by functionality.

The most commonly used library is the core library.

```
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%> - Core Library
<%@taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%> - 118N Capable Formatting Library
<%@taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql"%> - Relational Database Access Library
<%@taglib prefix="xml" uri="http://java.sun.com/jsp/jstl/xml"%> - XML Processing Library
```

JSTL Core Library Tags:

catch: Catches any Throwable that occurs in its body and optionally exposes it

choose: Simple conditional tag that establishes a context for mutually exclusive conditional operations, marked by <when> and <otherwise>

when: Subtag of <choose> that includes its body if its condition evaluates to 'true'

otherwise: Subtag of <choose> that follows <when> tags and runs only if all of the prior conditions evaluate to 'false'

if: Simple conditional tag, which evaluates its body if the supplied condition is true and optionally exposes a Boolean scripting variable representing the evaluation of this condition

import: Retrieves an absolute or relative URL and exposes its contents to either the page, a String in 'var', or a Reader in 'varReader'

forEach: The basic iteration tag, accepting many different collection types and supporting subsetting and other functionality

forTokens: Iterates over tokens, separated by the supplied delimiters

out: Like <%= ... >, but for expressions

param: Adds a parameter to a containing 'import' tag's URL

redirect: Redirects to a new URL

remove: Removes a scoped variable (from a particular scope, if specified)

set: Sets the result of an expression evaluation in a 'scope'

url: Creates a URL with optional query parameters

Create JSP Error Handlers

- A JSP page can designate another page to be used as its error handler.
- Error Handling pages can be designated using the `web.xml` deployment descriptor.
- Create Handler Page and declare it as an Error Page.
- Information about the error can be obtained from `requestScope` attributes.

```

some.jsp
<%@page errorPage="someErrorHandler.jsp"%>
...
someErrorHandler.jsp
<%@page isErrorPage="true"%>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
Origin Page: <c:out value="${requestScope['javax.servlet.error.request_uri']}'/>
Exception: <c:out value="${requestScope['javax.servlet.error.exception']}'/>
Exception Type: <c:out value="${requestScope['javax.servlet.error.exception_type']}'/>
Error Message: <c:out value="${requestScope['javax.servlet.error.message']}'/>
HTTP Status code: <c:out value="${requestScope['javax.servlet.error.status_code']}'/>

```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The `web.xml` file can be used to designate error handling pages for specific types of HTTP Response codes and JavaExceptions:

```

<error-page>
    <error-code>404</error-code>
    <location>/SomeErrorHandler.jsp</location>
</error-page>
<error-page>
    <exception-type>java.lang.Throwable</exception-type>
    <location>/AnotherErrorHandler.jsp</location>
</error-page>

```

Summary

In this lesson, you should have learned how to create Java Server Pages and how to:

- Describe JSP life cycle
- Describe JSP syntax
- Use Expression Language (EL)
- Use CDI Beans
- Use Tag Libraries
- Handle errors



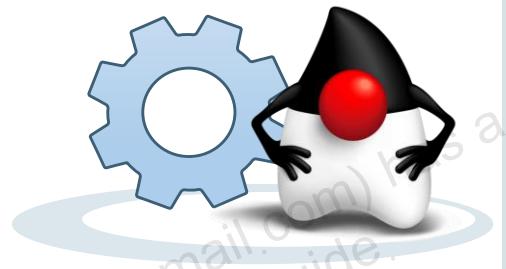
Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Practice Overview

This practice covers the following tasks:

- Modify ProductManager class to support Java Server Pages as a CDI bean
- Create ProductList JSP to display list of products
- Modify ProductList Servlet to execute product search on behalf of the ProductList JSP
- Create ProductEdit JSP to display product update form
- Create EditProduct WebFilter to intercept and handle requests for the EditProduct JSP



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Jairo Jose Silvera Sarmiento (jairo.silvera@gmail.com) has a
non-transferable license to use this Student Guide.

Implementing REST Services using JAX-RS API

9



ORACLE®

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Jairo Jose Silvera Sarmiento (jairo.silvera@gmail.com) has a
non-transferable license to use this Student Guide.

Objectives

This chapter teaches how to handle communications with REST Services

- Understand REST service conventions
- Create REST services using JAX-RS API
- Consume REST service within the client tier



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



REST Service Conventions and Resources

REST Service **Application** represents one or more **Resource**

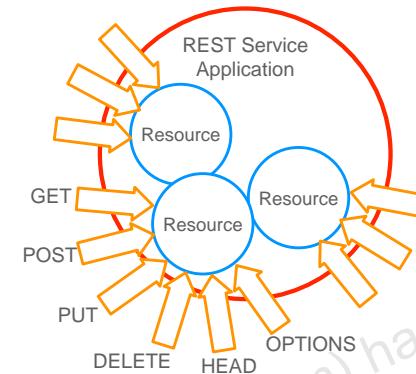
Each **Resource** represents a business entity and is mapped to its own URL

Each **Resource** defines a number of **operations** mapped to HTTP Methods

Typical conventional use of HTTP Methods:

- GET receives (queries) a collection of elements or a specific element identified by client
- POST creates new element
- PUT updates existing element
- DELETE removes an element
- Other operations are sometimes used to retrieve metadata

❖ Unlike SOAP, there is no standard for REST Services - they are considered to be a coding style rather than an actual protocol.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

REST is centered around an abstraction known as a "resource." Any named piece of information can be a resource.

A resource is identified by a uniform resource identifier (URI).

Clients request and submit representations of resources.

There can be many different representations available to represent the same resource.

A representation consists of data and metadata. – Metadata often takes the form of HTTP headers. Resources are interconnected by hyperlinks.

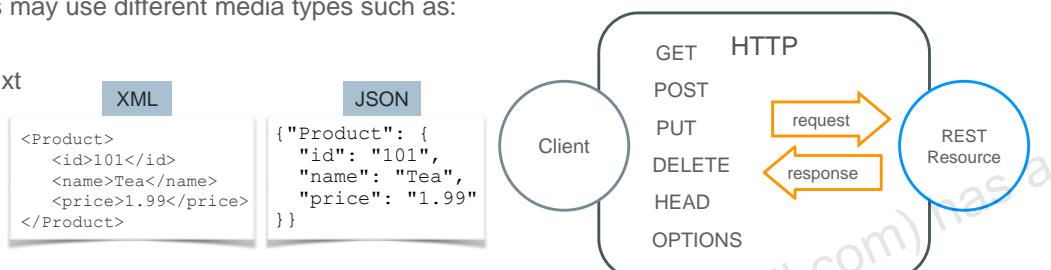
A RESTful web service is designed by identifying the resources. This is similar to Object Oriented Analysis and Design where you identify nouns in use-cases.

Unlike SOAP Services, there is no standard for REST Services. REST is considered to be a style of coding rather than an actual protocol.

REST Communication Model

REST Resources and REST Clients characteristics:

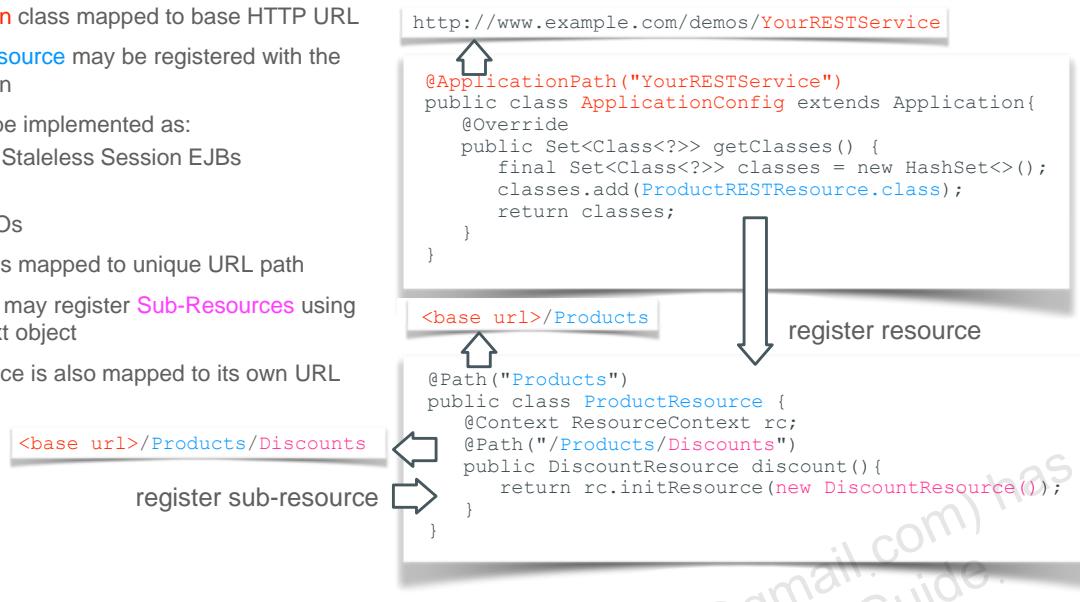
- REST Clients are often implemented by using JavaScript in Browser or Mobile Applications.
- Use HTTP as a transport layer and dispatch requests by using the GET, POST, PUT, DELETE, HEAD, and OPTIONS methods containing data that the client wants to pass to the REST Service.
- Clients handle responses that contain HTTP status codes and may also contain a body with server-generated content.
- Use of HTTP methods is conventional – REST services are not required to handle these in the same way.
- REST Resources may use different media types such as:
 - JSON
 - Plain Text
 - XML
 - And so on



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Implementing REST Services using JAX-RS API

- REST **Application** class mapped to base HTTP URL
- One or more **Resource** may be registered with the REST Application
- Resources can be implemented as:
 - Singleton or Stateless Session EJBs
 - CDI Beans
 - Simple POJOs
- Each Resource is mapped to unique URL path
- REST Resource may register **Sub-Resources** using ResourceContext object
- Each sub-resource is also mapped to its own URL



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

JAX-RS is a Java programming language API designed to make it easy to develop applications that use the REST architecture.

The JAX-RS API uses Java programming language annotations to simplify the development of RESTful web services. Developers decorate Java programming language class files with JAX-RS annotations to define resources and the actions that can be performed on those resources. JAX-RS annotations are runtime annotations; therefore, runtime reflection will generate the helper classes and artefacts for the resource. A Java EE application archive containing JAX-RS resource classes will have the resources configured, the helper classes and artefacts generated, and the resource exposed to clients by deploying the archive to a Java EE server.

A JAX-RS application consists of at least one resource class packaged within a WAR file. The base URI from which an application's resources respond to requests can be set one of two ways:

Using the `@ApplicationPath` annotation in a subclass of `javax.ws.rs.core.Application` packaged within the WAR. By default, all the resources in an archive will be processed for resources. Override the `getClasses` method to manually register the resource classes in the application with the JAX-RS runtime.

```

package demos.ws;

import java.util.Set;
import javax.ws.rs.core.Application;

```

```

@ApplicationPath("/YourRESTService")
public class MyApplication extends Application {
    @Override
    public Set<Class<?>> getClasses() {
        final Set<Class<?>> classes = new HashSet<>();
        classes.add(Products.class);
        return classes;
    }
    @Override
    public Set<Object> getSingletons() {
        return super.getSingletons();
    }
}

```

Or instead of the `@ApplicationPath` annotating use the `servlet-mapping` tag within the WAR's `web.xml` deployment descriptor

```

<servlet-mapping>
    <servlet-name>demos.ws.ApplicationConfig</servlet-name>
    <url-pattern>/YourRESTService/*</url-pattern>
</servlet-mapping>

```

All root resource classes (classes with `@Path("")` at the class level) will appear under the path value in the `@ApplicationPath("")` annotation. If you have only one root resource class, you can use `@Path("/")` to make the root resource available at the path in the `@ApplicationPath("resources")` annotation.

Root resource classes can be POJOs, CDI-managed beans, and stateless and singleton session beans. The JAX-RS implementation will obtain your root resource class from an Application subclass.

If the class is returned from `getClasses()`:

- JAX-RS will create an instance-per-request
- Perform a JAX-RS dependency inject
- Call any `@PostConstruct` methods
- Call the resource or resource-locator method

If the class is returned from `getSingletons()`:

- It is expected to be a singleton
- It is typically used for providers instead of resources

Mapping Resources to URI Paths

Path annotation identifies the URI path template to which the resource responds

- Can be specified at the class or method level
- Can restrict values using regular expressions
- Can handle multiple parameters
- Operations may Produce or Consume various Media Types typically JSON or XML

```
@Path("Products")
@Produces({MediaType.APPLICATION_JSON})
@Consumes({MediaType.APPLICATION_JSON})
public class ProductsRESTResource {
    @GET
    @Path("{code: [A-Z][0-9]+}")
    public Product findProduct(@PathParam("code") String code){...}
    @GET
    @Path("{from}/{to}")
    public List<Product> findRange(@PathParam("from") Integer from,
                                    @PathParam("to") Integer to){...}
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

By default, the URI variable must match the regular expression "[^]+?". This variable may be customized by specifying a different regular expression after the variable name. For example, if a user name must consist only of lowercase and uppercase alphanumeric characters, override the default regular expression in the variable definition: By default, the URI variable must match the regular expression "[^]+?". This variable may be customized by specifying a different regular expression after the variable name. If provided value does not match specified expression, a 404 (Not Found) response will be sent to the client.

Mapping REST Resource Operations

REST Resource defines operations handling different HTTP Method calls

- GET to read entities
- POST to create entities
- PUT to update entities
- DELETE to remove an entity
- HEAD to return headers
- OPTIONS to return metadata

```
@Path("Products")
@Produces({MediaType.APPLICATION_JSON})
@Consumes({MediaType.APPLICATION_JSON})
public class ProductsRESTResource {
    @GET
    public List<Products> findAll(){...}
    @GET @Path("{id}")
    public Product find(@PathParam("id") Integer id){...}
    @POST
    public void create(Product product){...}
    @PUT @Path("{id}")
    public void edit(@PathParam("id") Integer id, Product p){...}
    @DELETE @Path("{id}")
    public void remove(@PathParam("id") Integer id){...}
}
```

- ❖ POST can also be used to update an entity if id does not need to be set by the client
- ❖ PUT can also be used to create an entity with id set by the client
- ❖ JAX-RS API provides default implementation for HEAD and OPTIONS HTTP Methods



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

It is possible that some REST services may use POST method not only to create new, but also to update existing elements. However, it implies that client did not provide an element id. Therefore, server should be able assign new id to an element if it is creating it, or to determine which element to update without id supplied by the client. In a similar way PUT method can be used to create as well as update elements. When using PUT method client is supposed to provide an element id.

JAX-RS API provides default implementation for HEAD and OPTIONS HTTP Methods:

- HEAD by default invokes the implemented GET method (if present) and ignores the response entity (if set). This method can be used for obtaining metadata about the entity implied by the request without transferring the entity-body itself. It is often used for testing URL links for validity, accessibility, and recent modification. The response to a HEAD request may be cacheable in the sense that the information contained in the response may be used to update a previously cached entity from that resource. If the new field values indicate that the cached entity differs from the current entity (as would be indicated by a change in Content-Length, Content-MD5, ETag or Last-Modified), then the cache MUST treat the cache entry as stale.
- OPTIONS by default produce a list HTTP methods supported by the resource and return Web Application Definition Language (WADL)

Handling Different Media Types

REST Services may produce and consume information using various Media Types

- Handle various Media Types typically JSON or XML
- JAX-RS API auto-converts HTTP request and response message bodies to and from java objects using MessageBodyReader and MessageBodyWriter classes
- @Produces and @Consumes annotations can be set on a class or method level to describe Media Types that this service supports

```
@Stateless
@Path("Products")
@Produces({MediaType.APPLICATION_JSON})
public class ProductsRESTResource {
    @GET @Path("{id}")
    @Produces(MediaType.APPLICATION_XML)
    public List<Product> find(@PathParam("id") Integer id){...}
    @GET @Path("count")
    @Produces(MediaType.TEXT_PLAIN)
    public String countProducts(){...}
    @PUT @Path("{id}")
    @Consumes(MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON)
    public void edit(Product product){...}
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The @Produces and @Consumes annotations can be placed at the class level to define defaults for all class methods.

Adding the annotations at the method level overrides the class-level default. A method may both produce and consume content (depending on the HTTP method).

The @Produces and @Consumes annotations expect a string array for their value attribute:
`@Produces({"text/plain", "text/html"})`

Use MediaType constants to avoid typos: `@Produces({MediaType.TEXT_HTML, MediaType.TEXT_PLAIN})`

Internally, JAX-RS uses a MessageBodyReader and MessageBodyWriter to convert the HTTP body to and from a Java object.

- MessageBodyReader converts the HTTP body from the entity stream to a Java object (param method).
- MessageBodyWriter converts the HTTP body from a Java object to an entity stream (return method).

A RESTful service will typically produce and consume XML or JSON. XML support is provided by JAXB.

The supported entity types are:

- byte[] – All media types (/*)
- java.lang.String – All media types (/*)
- java.io.InputStream – All media types (/*)
- java.io.Reader – All media types (/*)
- java.io.File – All media types (/*)
- javax.activation.DataSource – All media types (/*)
- javax.xml.transform.Source – XML types (text/xml, application/xml and media types of the form application/*+xml)
- javax.xml.bind.JAXBElement and application-supplied JAXB classes XML types – (text/xml and application/xml and media types of the form application/*+xml)
- MultivaluedMap<String, String> – Form content (application/x-www-form-urlencoded)
- StreamingOutput – All media types (/*), MessageBodyWriter only
- java.lang.Boolean, java.lang.Character, java.lang.Number – Only for text/plain.
Corresponding primitive types supported via boxing/unboxing conversion.
- JsonStructure, JsonObject, and JSONArray – (application/json) On platforms that support JSON-P

Passing Parameters

REST Resource Handling classes may accept parameters in different forms

- Path Parameters
- Query Parameters
- Matrix Parameters
- HTTP Header values as Parameters
- Cookie values as Parameters
- As application/x-www-form-urlencoded request body

| | |
|----------------------------|-------------------|
| <base url>/Resource/acme | @PathParam("x") |
| <base url>/Resource?x=acme | @QueryParam("x") |
| <base url>/Resource;x=acme | @MatrixParam("x") |
| HTTP Header x=acme | @HeaderParam("x") |
| Cookie x=acme | @CookieParam("x") |

application/x-www-form-urlencoded x=acme @FormParam("x")

- Declarations of parameters typically appear as method parameters, constructor parameters, fields, or bean properties
- A Default value can be provided with any @*Param annotation, when the input element is missing

@DefaultValue(10)



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

You can combine any number of parameter types (including an unannotated body param) when declaring method arguments.

The @DefaultValue("value") annotation can be used in combination with the @*Param annotations to supply a default value when the input element is missing.

Declarations of parameters typically appear as method parameters of a resource method; however, they may also appear as constructor parameters, fields, or bean properties.

Validating Values

JAX-RS 2.0 supports the use of Bean Validation 1.1 (JSR-249)

- Validation constraints applied to resource method parameters, fields and property getters, as well as resource classes, entity parameters, and resource methods (return values)
- REST Services may use @Valid annotation to enable bean validations when representing an Entity that already uses bean validation annotations

```
@POST  
@PUT("{id}/{price}")  
@Consumes({MediaType.APPLICATION_JSON})  
public void updatePrice(@NotNull  
                        @PathParam("id") Integer id,  
                        @NotNull @DecimalMax(value="999.99")  
                        @DecimalMin(value="0.99")  
                        @PathParam("price") Integer price) {...}  
  
@POST  
@Consumes({MediaType.APPLICATION_JSON})  
public void create(@Valid Product product){...}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

You can use Bean Validation API with JAX-RS Services in exactly same way as you used it with JPA Entities

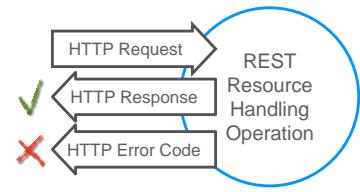
When a validation error occurs, a javax.validation.ValidationException or subclass is thrown. JAX-RS implementations must include an exception mapper, which maps a ValidationException to either a 400 (Bad Request) or 500 (Internal Server Error).

Handling Web Service Errors

When Handling Errors in REST Services

- Return HTTP status code to the client to indicate the nature of the error
- Operation may return a response containing an HTTP error code

```
ResponseBuilder rb = Response.status(Response.Status.NOT_FOUND);
return rb.build();
```



- Or throw a WebApplicationException to indicate HTTP error code to the client

```
throw new WebApplicationException(Response.Status.NOT_FOUND);
```

- Or throw one of the more specific exceptions (subclasses of the WebApplicationException)

```
throw new NotFoundException();
```

- Any other exceptions will require an ExceptionMapper (see notes)



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Status.NOT_FOUND represents an HTTP 404 error code. Therefore this code returns the same status:

```
ResponseBuilder rb = Response.status(404);
return rb.build();
or
throw new WebApplicationException(404);
or
throw new WebApplicationException(Response.Status.NOT_FOUND);
or
throw new NotFoundException();
```

NotFoundException is a subclass of the WebApplicationException. Other subclasses of the WebApplicationException class include:

| | | |
|------------------------------|-----|---|
| BadRequestException | 400 | Malformed message |
| NotAuthorizedException | 401 | Authentication failure |
| ForbiddenException | 403 | Not permitted to access |
| NotFoundException | 404 | Couldn't find resource |
| NotAllowedException | 405 | HTTP method not supported |
| NotAcceptableException | 406 | Client media type requested not supported |
| NotSupportedException | 415 | Client posted media type not supported |
| InternalServerErrorException | 500 | General server error |
| ServiceUnavailableException | 503 | Server is temporarily unavailable or busy |

Automatic mapping of Java Exceptions to Response objects is defined by the ExceptionMapper interface.

Providers implementing ExceptionMapper contract must be either programmatically registered in a JAX-RS runtime or must be annotated with @Provider annotation to be automatically discovered by the JAX-RS runtime during a provider scanning phase. It defines toResponse(E exception) operation that maps an Exception to a Response. Returning null results in a Response.Status.NO_CONTENT response. Throwing a runtime exception results in a Response.Status.INTERNAL_SERVER_ERROR response.

If in your application you have designed a custom exception (ProductNotFoundException), then it should be mapped to specific HTTP response code using a mapper:

```
@Provider
public class ProductNotFoundMapper
    implements ExceptionMapper<ProductNotFoundException> {

    public Response toResponse(ProductNotFoundException e) {
        return Response.status(Response.Status.NOT_FOUND).build();
    }
}
```

Asynchronous REST Services

JAX-RS Web Services can process requests asynchronously

- AsyncResponse is similar to the Servlet 3.0 AsyncContext class and allows asynchronous request handling
- AsyncResponse resume operation produces the response to the client
- TimeoutHandler provides a mechanism for defining custom resolution of time-out events

```
@Path("/some")
public class AsyncService {
    @GET
    public void doThings(@Suspended final AsyncResponse ar) {
        ar.setTimeoutHandler(new TimeoutHandler() {
            public void handleTimeout(AsyncResponse ar {
                ar.resume(Response.status(
                    Response.Status.SERVICE_UNAVAILABLE)
                    .entity("Operation time out.").build()));
            }
        });
        ar.setTimeout(20, TimeUnit.SECONDS);
        new Thread(new Runnable(){
            public void run() {
                Object result = ...
                ar.resume(result);
            }}).start();
    }
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Suspended annotation injects a suspended AsyncResponse into a parameter of an invoked JAX-RS resource or sub-resource method.

The injected AsyncResponse instance is bound to the processing of the active request and can be used to resume the request processing when a response is available.

By default there is no suspend timeout set and the asynchronous response is suspended indefinitely. The suspend timeout as well as a custom timeout handler can be specified programmatically using the AsyncResponse.setTimeout(long, TimeUnit) and AsyncResponse.setTimeoutHandler(TimeoutHandler) methods.

AsyncResponse has following capabilities:

Produce response that is either a normal response or an error with the following pair of methods:

- resume(Object response) - Resume the suspended request processing using the provided response data
- resume(Throwable response) - Resume the suspended request processing using the provided throwable

Cancel production of Response

- **cancel()** - Cancel the suspended request processing
- **cancel(Date retryAfter)** - Cancel the suspended request processing and set Retry-After header as a point in time
- **cancel(int retryAfter)** - Cancel the suspended request processing and set Retry-After header as a period of time in seconds

Set timeout and timeout handler

- **setTimeout(long time, TimeUnit unit)** - Set/update the suspend timeout.
- **setTimeoutHandler(TimeoutHandler handler)** - Set/replace a time-out handler for the suspended asynchronous response.

Find out status of the Response

- **isCancelled()** - Check if the asynchronous response instance has been cancelled
- **isDone()** - Check if the processing of a request this asynchronous response instance belongs to has finished
- **isSuspended()** - Check if the asynchronous response instance is in a suspended state

Register asynchronous processing lifecycle callback classes to receive lifecycle events for the asynchronous response based on the implemented callback interfaces

- register(Class<?> callback, Class<?>... callbacks)
- register(Object callback)
- register(Object callback, Object... callbacks)

CompletionCallback interface defines onComplete(Throwable throwable) operation that describes a completion callback notification method that will be invoked when the request processing is finished, after a response is processed and is sent back to the client or when an unmapped throwable has been propagated to the hosting I/O container.

An unmapped throwable is propagated to the hosting I/O container in case no exception mapper has been found for a throwable indicating a request processing failure. In this case a non-null unmapped throwable instance is passed to the method. Note that the throwable instance represents the actual unmapped exception thrown during the request processing, before it has been wrapped into an I/O container-specific exception that was used to propagate the throwable to the hosting I/O container. If Throwable parameter is null it indicates that the request processing has been completed and a response that has been sent to the client.

Asynchronous EJB and REST Services

Combination of EJB's @javax.ejb.Asynchronous annotation and the @Suspended AsyncResponse enables asynchronous execution of business logic with eventual notification of the interested client.

```
@Stateless  
@Path("products")  
public class ProductsRESTResource {  
    @POST  
    @Asynchronous  
    public void handle(@Suspended AsyncResponse ar, Product p)  
        // business logic execution  
        Response response = Response.ok(p).build();  
        ar.resume(response);  
    }  
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Invoking REST Service from JavaScript Client

JavaScript may invoke REST Service using Asynchronous JavaScript and XML API

- AJAX API handles both Synchronous and Asynchronous HTTP communications
- XMLHttpRequest object represents all types of http requests, regardless if they are actually xml or not
- Request URL pointing to the REST Service resource is prepared and opened
- Request is dispatched to the server via send operation, that can take an object as an argument
- When response has been received the onreadystatechange function is invoked
- Different other JavaScript APIs are available to automate and simplify creation of such clients

```
var request = new XMLHttpRequest();
var id = document.getElementById("productId").value;
var url = "http://www.example.com/demos/YourRESTService/Products/" + id;
request.open('GET', url, true);
request.send();
request.onreadystatechange = function() {
    if (request.readyState == 4) {
        if (request.status == 200) {
            var product = JSON.parse(request.responseText);
            // handle product object
        } else {
            alert("Error: " + request.responseText);
        }
    }
}
```

- ❖ For more information on JavaScript and how it is used to invoke REST Services refer to "JavaScript and HTML5: Develop Web Applications" training course



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Invoking REST Service from Java Client

Java clients may invoke REST Services using JAX-RS Jersey reference implementation library

- `ClientBuilder` class creates a `Client` object that handles a client-side communication infrastructure
- `Client` object must be properly closed before being disposed to avoid leaking resources.
- `WebTarget` object represents a REST Service resource
- `WebTarget` points to the resource via the `HTTP URL`, which can be constructed out of `Path components`
- JAX-RS API performs `conversions` between REST messages (such as JSON, XML etc.) and Java Objects
- `WebTarget` allows to `submit GET/POST/PUT/DELETE etc. requests`

```
String baseUrl = "http://www.example.com/demos/YourRESTService/";
Client client = ClientBuilder.newClient();
WebTarget webTarget = client.target(baseUrl).path("Products").path(id.toString());
Response response =
webTarget.register(Product.class).request(MediaType.APPLICATION_JSON).buildGet().invoke();
Product product = response.readEntity(Product.class);
client.close();
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Invoking REST Service from Asynchronous Java Client

Java clients may invoke REST Services Asynchronously

- Prepare and initialize Client and WebTarget objects
- Prepare `AsyncInvoker` object using `async` method
- Register `InvocationCallback` listener and override `completed` and `failed` methods
- Handle service `responses` or `faults` within these methods

```
Client client = ...
WebTarget target = ...
target.register(Product.class).request(MediaType.APPLICATION_JSON)
    .async()
    .get(
        new InvocationCallback<Product>() {
            @Override
            public void completed(Response response) {...}
            @Override
            public void failed(Throwable fault) {...}
        });
});
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Summary

In this lesson you should have learned how to handle communications with REST Services

- Understand REST service conventions
- Create REST services using JAX-RS API
- Consume REST service within the client tier



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



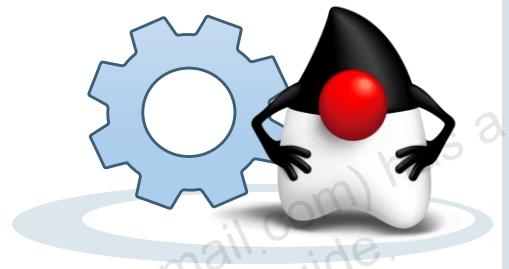
Practice

This practice covers the following tasks:

- Create REST Service to check if discount is available for a product
- Add JavaScript code to your web application to invoke Discount REST Service
- Create Java Client Application to invoke Discount REST Service



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Creating Java Applications with WebSockets

10



ORACLE®

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Jairo Jose Silvera Sarmiento (jairo.silvera@gmail.com) has a
non-transferable license to use this Student Guide.

Objectives

After completing the lesson, you should be able to:

- Explain WebSockets communication style
- Create WebSocket Endpoint Handlers using JSR 356 API
- Manage WebSocket Endpoint life cycle
- Produce and consume WebSocket messages
- Handle errors
- Encode and decode JSON messages
- Provide WebSocket Client Endpoint handler using Java and JavaScript



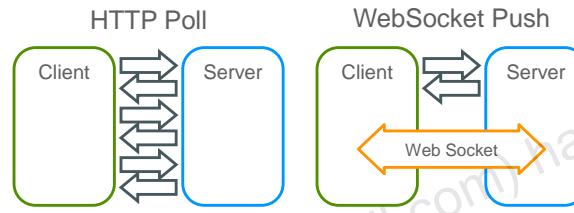
Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



WebSockets Network Protocol

WebSocket protocol supports bidirectional communication between client and server.

- Unlike WebSocket, HTTP Protocol only allows clients to be originators of calls.
- WebSocket connection starts with an HTTP handshake but does not use HTTP after that.
- Once a WebSocket connection is established, it allows either side (client or server) to send or receive messages in any order, enabling server to push information to the client.
- To support a WebSocket connection, the server can provide one instance of endpoint handler for the life of this connection, making application stateful in a very similar way to that of an HTTP Session.
- WebSocket clients are typically implemented using JavaScript.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The WebSocket Protocol is defined with RFC6455 standard: <http://tools.ietf.org/html/rfc6455>

WebSocket communications are supported by:

- JavaScript Client API defined by the W3C <http://www.w3.org/TR/websockets/>
- Java Client and Server API defined by JSR 356 <https://www.jcp.org/en/jsr/detail?id=356>

Traditional HTTP communication model assumes that each call is originated by the client. So if client wants to get some new or updates information from the server, such client might have to poll server for this information. This means client has to perform multiple calls in a loop just to see if server has any new or updated information for this client.

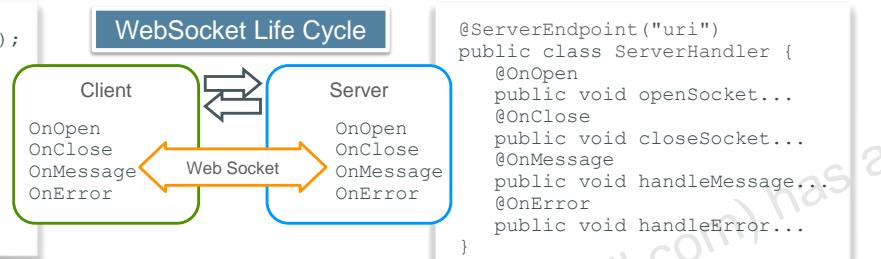
Unlike the traditional HTTP communication model, where the client is always an originator of every request, WebSockets treats client and server as endpoints with same functionality. This means that once WebSocket connections is established, both server as well as the client can originate calls using this WebSocket connection, enabling server to push information to the client.

WebSocket Life Cycle

WebSocket defines Client and Server communication endpoints handlers.

- Both Client and Server endpoints provide symmetrical capabilities.
- They are defined by the following operations:
 - OnOpen - is invoked when a new WebSocket connection is opened
 - OnClose - is invoked when a new WebSocket connection is closed
 - OnMessage - is invoked every time a message arrives from Server or from Client via the opened socket
 - OnError - is invoked when errors occurred when handling socket communications
- This example illustrates JavaScript Client and Java Server Endpoint handlers

```
var socket = new WebSocket("uri");
socket.onopen
  = function(event){...}
socket.onclose
  = function(event){...}
socket.onmessage
  = function(event){...}
socket.onerror
  = function(event){...}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

WebSocket Client and Server Endpoint handlers can be implemented in various programming languages. However, typically Client Endpoints are implemented in JavaScript and Server Endpoints in Java.

Defining WebSocket Endpoints

WebSocket server endpoints are annotated and packaged in a WAR.

- ServerEndpoint class is mapped to WebSocket url, possibly with parameters.
- Both @ClientEndpoint and @ServerEndpoint classes contain methods that respond to various events:
 - @OnOpen may accept EndpointConfig and custom parameters.
 - @OnClose may accept CloseReason and custom parameters.
 - @OnMessage must accept text or binary message and may accept custom parameters and may return a value.
 - @OnError must accept Throwable and may accept custom parameters.
- All these event handler methods may optionally accept WebSocket Session Object.

```
ws://www.example.com/demos/order
  ↗ @ServerEndpoint("/order")
    public class OrderServerWebSocketHandler {
        @OnOpen public void openSocket(Session session) {...}
        @OnClose public void closeSocket(CloseReason reason) {...}
        @OnMessage public void handleMessage(String message) {...}
        @OnError public void handleError(Throwable error) {...}
    }
```


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

ServerEndpoint is a class level annotation that decorates this class as a web socket endpoint that will be deployed and made available in the URI-space of a web socket server. The annotation allows the developer to define the URL (or URI template) at which this endpoint will be published and other important properties of the endpoint to the websocket runtime, such as the encoders that it uses to send messages. The ServerEndpoint annotated class must have a public no-arg constructor.

The OnOpen method level annotation is used to decorate a Java method that wishes to be called when a new web socket session is open.

The method may only take the following parameters:

- Session (optional parameter)
- EndpointConfig (optional parameter)
- Zero to more custom parameters annotated with the PathParam annotation

The OnClose method level annotation is used to decorate a Java method that wishes to be called when a new web socket session is closing.

The method may only take the following parameters:

- Session (optional parameter)
- CloseReason (optional parameter)
- Zero to more custom parameters annotated with the PathParam annotation

The `OnError` method level annotation is used to decorate a Java method that wishes to be called in order to handle errors.

The method may only take the following parameters:

- Session (optional parameter)
- Throwable - must be present
- Zero to more custom parameters annotated with the `PathParam` annotation

The `OnMessage` method level annotation can be used to make a Java method receive incoming web socket messages. Each websocket endpoint may only have one message handling method for each of the native websocket message formats:

- Text – can be represented by `java String` or `String` and `Boolean` parameter to receive message in parts, or `Reader`, or other classes that can be converted to and from `String` form using `Decoder.Text` or `Decoder.TextStream`.
- Binary – can be represented by `java byte[]` or `ByteBuffer`, or `byte[]` or `ByteBuffer` and `Boolean` parameter to receive message in parts, or `InputStream`, or other classes that can be converted to and from binary form using `Decoder.Binary` or `Decoder.BinaryStream`.
- Pong – `PongMessage` that represent a response to a Ping and is used to check if connection is still alive.

On Message method parameters and return values are described later in the lesson in more detail.

The `PathParam` annotation may be used to annotate method parameters on server endpoints where a URI-template has been used in the path-mapping of the `ServerEndpoint` annotation. The method parameter may be of type `String`, any Java primitive type or any boxed version thereof. If a client URI matches the URI-template, but the requested path parameter cannot be decoded, then the websocket's error handler will be called.

The `EndpointConfig` object contains all the information needed during the handshake process for this end point. All endpoints specify, for example, a URI. In the case of a server endpoint, the URI signifies the URI to which the endpoint will be mapped. In the case of a client application, the URI signifies the URI of the server to which the client endpoint will attempt to connect.

Using PathParam Annotation

The PathParam annotation maps WebSocket operation parameters to URL.

- Such parameters form part of the URI template for the given WebSocket Server endpoint.
- WebSocket operations may access such parameters.
- Different other values may be passed between the client and server endpoints OnMessage operations.



```
ws://www.example.com/demos/order/101
```

```
↑  
@ServerEndpoint("/order/{id}")  
public class OrderServerWebSocketHandler {  
    @OnOpen  
    public void openSocket(Session session, @PathParam("id") int id){...}  
    @OnClose  
    public void closeSocket(CloseReason reason, @PathParam("id") int id){...}  
    @OnMessage  
    public void handleMessage(String message, @PathParam("id") int id){...}  
    @OnError  
    public void handleError(Throwable error, @PathParam("id") int id){...}  
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The PathParam annotation may be used to annotate method parameters on server endpoints where a URI-template has been used in the path-mapping of the ServerEndpoint annotation. The method parameter may be of type String, any Java primitive type or any boxed version thereof. If a client URI matches the URI-template, but the requested path parameter cannot be decoded, then the websocket's error handler will be called.

Using WebSocket Session

A WebSocket session represents a conversation between two web socket endpoints.

A WebSocket session allows to:

- Find session status with the isOpen method
- Register one of each type of message handlers to handle incoming messages for this session:
 - Text Message Handler
 - Binary Message Handler
 - Pong Message Handler
- Close session when the conversation between client and server is over
 - Session closed with no parameters defaults to NORMAL_CLOSURE and no reason message
- Access security information
- Access set of other sessions related to the same endpoint to implement cross-session conversations

```
session.isOpen();
session.addMessageHandler(...);
session.setMaxIdleTimeout(...);
Principal principal = session.getUserPrincipal();
Set<Session> allSessions = session.getOpenSessions();
session.close(CloseReason.CloseCodes.TRY AGAIN LATER);
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

A Web Socket session represents a conversation between two web socket endpoints. As soon as the websocket handshake completes successfully, the web socket implementation provides the endpoint an open websocket session. The endpoint can then register interest in incoming messages that are part of this newly created session by providing a MessageHandler to the session, and can send messages to the other end of the conversation by means of the RemoteEndpoint object obtained from this session.

Method close() terminates the connection and no data can be transferred until the connection between client and server is reestablished. Invoking this method also means a "goodbye" hand shake.

Once the session is closed, it is no longer valid for use by applications. Calling any of its methods (with the exception of the close() methods) once the session has been closed will result in an IllegalStateException being thrown. Developers should retrieve any information from the session during the Endpoint.onClose(javax.websocket.Session, javax.websocket.CloseReason) method. Following the convention of Closeable calling the Session close() methods after the Session has been closed has no effect.

Session objects may be called by multiple threads. Implementations must ensure the integrity of the mutable properties of the session under such circumstances.

A maximum of one message handler per native websocket message type (text, binary, pong) may be added to each Session, that is, a maximum of one message handler to handle incoming text messages and a maximum of one message handler for handling incoming binary messages, and a maximum of one for handling incoming pong messages. For further details of which message handlers handle which of the native websocket message types please see MessageHandler.Whole and MessageHandler.Partial. Adding more than one of any one type will result in a runtime exception.

For more information see: <http://docs.oracle.com/javaee/7/api/javax/websocket/Session.html>

For a full set of session close reason codes refer to:

<http://docs.oracle.com/javaee/7/api/javax/websocket/CloseReason.CloseCodes.html>

Using RemoteEndpoint Objects

RemoteEndpoint Objects are used to send messages to the session counterpart.

- RemoteEndpoint represents a ClientEndpoint on a server or ServerEndpoint on a client.
- RemoteEndpoints could be of two types:
 - Synchronous
 - Asynchronous

```
RemoteEndpoint.Basic br = session.getBasicRemote();
br.sendObject(...);
```

```
RemoteEndpoint.Async ar = session.getAsyncRemote();
Future<Void> transmissionStatus = ar.sendObject(...);
```

- RemoteEndpoints are used to implement communications at any phase of a WebSocket lifecycle within the OnOpen, OnClose, OnMessage, and OnError operations.
- The onMessage operation may simply return a value if response is produced as a direct result of a received call.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

RemoteEndpoint.Basic interface is the representation of the peer of a web socket conversation that has the ability to send messages synchronously. The point of completion of the send is defined when all the supplied data has been written to the underlying connection. The methods for sending messages on RemoteEndpoint.Basic block until this point of completion is reached, except for getSendStream and getSendWriter, which present traditional blocking I/O streams to write messages.

If the WebSocket connection underlying this RemoteEndpoint is busy sending a message when a call is made to send another one, for example if two threads attempt to call a send method concurrently, or if a developer attempts to send a new message while in the middle of sending an existing one, the send method called while the connection is already busy may throw an IllegalStateException.

RemoteEndpoint.Async interface is the representation of the peer of a web socket conversation that has the ability to send messages asynchronously. The point of completion of the send is defined when all the supplied data has been written to the underlying connection. The completion handlers for the asynchronous methods are always called with a different thread from that which initiated the send.

To operate on with asynchronous transmissions of messages RemoteEndpoint.Async methods return before they complete transmission of messages. Developers use the returned Future object to track progress of the transmission. The Future's get() method returns null upon successful completion. Errors in transmission are wrapped in the ExecutionException thrown when querying the Future object.

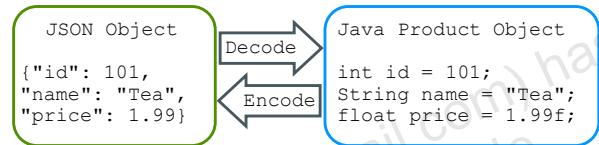
Both of the RemoteEndpoint variants support different ways of sending messages, using different versions of the operation send() to send text files, binary data or images, text messages and so on.

Encode and Decode Messages

Messages that travel between server and client endpoints must be converted to and from Java.

- JavaScript Object Notation (JSON) is commonly used by RESTful web services and WebSocket applications.
- Java API for JSON Processing (JSON-P) JSR 353 is used to perform JSON message encoding and decoding.
- Streaming JSON-P classes:
 - JsonParser – A pull parser for reading JSON data
 - JsonGenerator – A JSON generator that uses method chaining
- Object-based JSON-P classes:
 - JsonReader – Reads from an InputStream and produces an object graph
 - JsonWriter – Writes a JSON-P-specific object graph to an OutputStream
- JsonObject class represents JSON Object in Java.
 - Messages can be encoded and decoded within server and client endpoint classes, or by separate classes, registered with endpoint handlers.

- ❖ For more information on JavaScript and how it is used to implement WebSocket communications, refer to "JavaScript and HTML5: Develop Web Applications" training course.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Third-party Java JSON libraries, such as Gson, have been around for years. However, JSON APIs are now starting to become the official components of the Java platform by using the JEP/JSR process. JSON-P was released as part of Java EE 7 and several other JSON standards are under development:

- **JSR 367: Java API for JSON Binding (JSON-B):** This API will standardize JSON to Java binding similar to what JAXB does for XML. It is currently scheduled for release as a Java EE 8 component. For more, see <https://jcp.org/en/jsr/detail?id=367>.
- **JEP 198: Light-Weight JSON API:** This API provides a lightweight JSON library under java.util and is under consideration for Java SE 9. For more, see <http://openjdk.java.net/jeps/198>.

The streaming classes parse or generate JSON as the data streams through your application eliminating the need to have all of the JSON data in memory as objects.

The object-based JSON reader or writer classes use JSON-P class types to represent JSON constructs.

- **JsonStructure** – The common JSON object supertype
- **JsonObject** – A JSON object that uses string keys to uniquely identify values
- **JsonArray** – A JSON object that uses int index values to uniquely identify values

Handle WebSocket Messages

The OnMessage operation handles client calls within a server or server calls within a client endpoint.

- Must accept text or binary message and may accept custom parameters
- May accept additional parameters mapped with PathParam annotation
- May accept Session parameter
- May be void or may return a value

```
@OnMessage
public String findProduct(String value) {
    JsonObject inObj = Json.createReader(new StringReader(value)).readObject();
    int id = inObj.getInt("id");
    Product product = pm.findProduct(id);
    if (product == null) {
        throw new RuntimeException("Product with id " + id + " not found");
    }
    JsonObject outObj = Json.createObjectBuilder().add("id", product.getId())
        .add("name", product.getName())
        .add("price", product.getPrice()).build();
    StringWriter stringWriter = new StringWriter();
    JsonWriter writer = Json.createWriter(stringWriter);
    writer.writeObject(outObj);
    writer.close();
    return stringWriter.getBuffer().toString();
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The OnMessage method may have a non-void return type, in which case the web socket runtime must interpret this as a web socket message to return to the peer. The allowed data types for this return type, other than void, are String, ByteBuffer, byte[], any Java primitive or class equivalent, and anything for which there is an encoder. If the method uses a Java primitive as a return value, the implementation must construct the text message to send using the standard Java string representation of the Java primitive unless there developer provided encoder for the type configured for this endpoint, in which case that encoder must be used. If the method uses a class equivalent of a Java primitive as a return value, the implementation must construct the text message from the Java primitive equivalent as described above.

Note that if developer closes the session during the invocation of a method with a return type, the method will complete but the return value will not be delivered to the remote endpoint. The send failure will be passed back into the endpoint's error handling method.

The allowed parameters could be one of any of the following choices:

- String to receive the whole message
- Java primitive or class equivalent to receive the whole message converted to that type
- String and Boolean pair to receive the message in parts
- Reader to receive the whole message as a blocking stream
- Any object parameter for which the endpoint has a text decoder (Decoder.Text or Decoder.TextStream)
- byte[] or ByteBuffer to receive the whole message
- byte[] and Boolean pair, or ByteBuffer and Boolean pair to receive the message in parts

- InputStream to receive the whole message as a blocking stream
- Any object parameter for which the endpoint has a binary decoder (Decoder.Binary or Decoder.BinaryStream)
- PongMessage for handling pong messages - as a response to a ping

There could be zero or more String or Java primitive, or corresponding wrapper type parameters annotated with the PathParam annotation for server endpoints and an optional Session parameter.

The parameters may be listed in any order.

Handle WebSocket Errors

The OnError operation handles exceptions produced by other WebSocket operations.

- Must accept Throwable parameter
- May accept additional parameters mapped with PathParam annotation
- May accept Session parameter
- RemoteEndpoint object used to dispatch error messages to the WebSocket counterpart

```
@OnError
public void handleError(Session session, Throwable exception) {
    RemoteEndpoint.Basic remote = session.getBasicRemote();
    try {
        remote.sendObject(exception.getMessage());
    } catch (IOException ex | EncodeException ex) {
        Logger.getLogger(WebSocketServer.class.getName()).log(Level.SEVERE, null, ex);
        throw new RuntimeException("Error sending product", ex);
    }
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Encoding and Decoding WebSocket Messages

Encodes and Decoders convert Java Objects to and from WebSocket operable formats such as JSON.

- OnMessage operation or RemoteEndpoint objects are allowed to send or receive custom Java Objects only if the appropriate encoders and decodes are registered.

```
@ServerEndpoint(value="/order")
public class ProductServerSocketHandler {
    @OnMessage
    public String updatePrice(String product) {
        ...
        product.setPrice(2.99f); // use Java object
        ...
        return jsonText;
    }
    ...
}

@ServerEndpoint(value="/order", encoders={Product2JSON.class}, decoders={JSON2Product.class})
public class ProductServerSocketHandler {
    @OnMessage
    public Product updatePrice(Product product) {
        product.setPrice(2.99f); // just use Java object
        return product;
    }
    ...
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Implementing WebSocket Message Encoder

Encoders are used to convert Java Objects into WebSocket writeable format such as JSON.

- Override encode method:
 - Accept Java Object that requires conversion
 - Return conversion result such as JSON String
- Depending on required conversion result type implement one of the following interfaces:
 - Encoder.Text
 - Encoder.TextStream
 - Encoder.Binary
 - Encoder.BinaryStream

```
public class Product2JSON implements Encoder.Text<Product> {  
    @Override public String encode(Product product)  
        throws EncodeException {  
        JsonObject obj = Json.createObjectBuilder()  
            .add("id", product.getId())  
            .add("name", product.getName())  
            .add("price", product.getPrice()).build();  
        StringWriter stringWriter = new StringWriter();  
        Json.createWriter(stringWriter).writeObject(product).close();  
        return stringWriter.getBuffer().toString();  
    }  
    @Override public void init(EndpointConfig config) {}  
    @Override public void destroy() {}  
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

- Implementations of Encoder.Text interface override this operation:
 - String encode(T object) - Encode the given object into a String.
- Implementations of Encoder.Binary interface override this operation:
 - ByteBuffer encode(T object) - Encode the given object into a byte array
- Implementations of Encoder.TextStream interface override this operation:
 - void encode(T object, Writer writer) - Encode the given object to a character stream writing it to the supplied Writer.
- Implementations of Encoder.BinaryStream interface override this operation:
 - void encode(T object, OutputStream os) - Encode the given object into a binary stream written to the implementation provided by the OutputStream.

Implementing WebSocket Message Decoder

Decoders are used to convert WebSocket operable format such as JSON into Java Objects.

- Override decode method
 - Accept value that requires conversion
 - Return converted Java Object
- Depending on accepted value type implement one of the following interfaces:
 - Decoder.Text
 - Decoder.TextStream
 - Decoder.Binary
 - Decoder.BinaryStream

```
public class JSON2Product implements Decoder.Text<Product> {
    @Override public Product decode(String message)
        throws DecodeException {
        JsonObject obj = Json.createReader(
            new StringReader(message)).readObject();
        int id = obj.getInt("id");
        String name = obj.getString("name");
        float price = (float)(obj.getJsonNumber("price").doubleValue());
        return new Product(id, name, price);
    }
    @Override public boolean willDecode(String message) {
        return message != null; //can write message validation here
    }
    @Override public void init(EndpointConfig config) {}
    @Override public void destroy() {}
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

- Implementations of Decoder.Text interface override following operations:
 - T decode(String s) - Decode the given String into an object of type T.
 - boolean willDecode(String s) - Answer whether the given String can be decoded into an object of type T.
- Implementations of Decoder.Binary interface override following operations:
 - T decode(ByteBuffer bytes) - Decode the given bytes into an object of type T.
 - boolean willDecode(ByteBuffer bytes) - Answer whether the given bytes can be decoded into an object of type T.
- Implementations of Decoder.TextStream interface override this operation:
 - T decode(Reader reader) - Read the websocket message from the implementation provided Reader and decodes it into an instance of the supplied object type.
- Implementations of Decoder.Text interface override this operation:
 - T decode(InputStream is) - Decode the given bytes read from the input stream into an object of type T.

Creating JSON Messages

When implementing JSON Encoder, choose between:

- JsonGenerator, builds JSON text and writes it to an OutputStream or a Writer

Writing data directly to stream

```
JsonGenerator jgen = Json.createGenerator(<target stream>);  
jgen.writeStartObject().write("id", "101")  
    .write("name", "Tea").writeEnd();  
jgen.flush();
```

- JsonWriter, writes that object graph as JSON text to an OutputStream or a Writer

A JsonObjectBuilder allows to build an object graph of JSON-specific objects.

```
JsonWriter jw = Json.createWriter(<target stream>);  
JsonObject obj = Json.createObjectBuilder()  
    .add("id", "101")  
    .add("name", "Tea").build();  
jw.write(obj);
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Parsing JSON Messages

When implementing JSON Decoder choose between:

- A JsonParser converts JSON text into a sequence of events.

It is a fast, low-level method for parsing JSON without loading the JSON data into an object graph.

```
JsonParser parser = Json.createParser(<input stream>);
while(parser.hasNext()) {
    Event e = parser.next();
    switch(e) {
        case JsonParser.Event.KEY_NAME:
            String elementName = parser.getString();
        case JsonParser.Event.VALUE_STRING:
            String elementValue = parser.getString();
        case JsonParser.Event.VALUE_NUMBER:
            int numberValue = parser.getInt();
    }
}
```

- A JsonReader converts JSON text into an object graph.

Convenient to use for smaller objects

```
JsonReader reader = Json.createReader(<input stream>);
JsonObject obj = reader.readObject();
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

JsonParser.Event types are:

- START_ARRAY - Start of a JSON array
- START_OBJECT - Start of a JSON object
- END_ARRAY - End of a JSON array
- END_OBJECT - End of a JSON object
- KEY_NAME - Name in a name/value pair of a JSON object
- VALUE_FALSE - False value in a JSON array or object
- VALUE_TRUE - True value in a JSON array or object
- VALUE_NULL - Null value in a JSON array or object
- VALUE_NUMBER - Number value in a JSON array or object
- VALUE_STRING - String value in a JSON array or object

Invoking WebSocket Server from a JavaScript Client

JavaScript may invoke WebSocket Server Endpoint handler and receive callbacks from it.

- New WebSocket object is opened pointing to the WebSocket Server Endpoint.
- Web Socket Client Endpoint life-cycle operations are registered with the socket.
 - onmessage - Handles messages received from the server
 - onerror - Handles WebSocket errors
 - onopen - Executes custom code when socket is opened
 - onclose - Executes custom code when socket is closed
- Use send operation to dispatch messages to the WebSocket Server Endpoint.

```
var socket = new WebSocket("ws://www.example.com/demos/product");
socket.onmessage = function (event) {
    var product = JSON.parse(event.data);
    // handle product object
}
socket.onerror = function(event){ alert(event.data); }
function findProduct() {
    var productId = document.getElementById("pid").value;
    socket.send(productId);
}
```

- ❖ For more information on JavaScript and how it is used to implement WebSocket communications, refer to "JavaScript and HTML5: Develop Web Applications" training course.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Invoking WebSocket Server from a Java Client

Java Application can implement a WebSocket Client Endpoint handler.

- WebSocket **Client Endpoint** mirrors the Server Endpoint, providing OnOpen, OnClose, OnError and OnMessage operations
- WebSocket Client:
 - **Connects to WebSocket Server**
 - Registers **Client Endpoint Handler**
 - Acquires **RemoteEndpoint**
 - Prepares and dispatches messages
 - **Closes** WebSocket session

```
@ClientEndpoint  
public class SocketHandler {  
    @OnOpen  
    public void onOpen(Session session) {...}  
    @OnClose  
    public void onClose(Session session, CloseReason reason) {...}  
    @OnError  
    public void onError(Session session, Throwable ex) {...}  
    @OnMessage  
    public void onMessage(String message) {...}
```

```
URI uri = new URI("ws://www.example.com/demos/order");  
WebSocketContainer container = ContainerProvider.getWebSocketContainer();  
Session session = container.connectToServer(new SocketHandler(), uri);  
RemoteEndpoint.Async remote = session.getAsyncRemote();  
Product product = new Product();  
...  
remote.sendObject(product);  
session.close(new CloseReason(CloseReason.CloseCodes.NORMAL_CLOSURE, "Goodbye"));
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Summary

In this lesson, you should have learned how to:

- Explain WebSockets communication style
- Create WebSocket Endpoint Handlers using JSR 356 API
- Manage WebSocket Endpoint life cycle
- Produce and consume WebSocket messages
- Handle errors
- Encode and decode JSON messages
- Provide WebSocket Client Endpoint handler using Java and JavaScript



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Practice

This practice covers the following tasks:

- Creating WebSocket Chat Server to allow callers to exchange chat messages
- Creating HTML and JavaScript clients to interact with the WebSocket Chat Server
- Creating a Java Client Application to interact with the WebSocket Chat Server



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Jairo Jose Silvera Sarmiento (jairo.silvera@gmail.com) has a
non-transferable license to use this Student Guide.

Developing Web Applications Using JavaServer Faces

11



ORACLE®

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Jairo Jose Silvera Sarmiento (jairo.silvera@gmail.com) has a
non-transferable license to use this Student Guide.

Objectives

This lesson shows how to develop web applications using JavaServer Faces

- Describe JSF lifecycle and architecture
- Understand JSF syntax
- Use JSF Component Libraries
- Apply Validators and Converters to UIComponents
- Use UI templates
- Define navigation
- Handle localization
- Produce messages
- Use Expression Language (EL)
- Use CDI Beans
- Add AJAX support



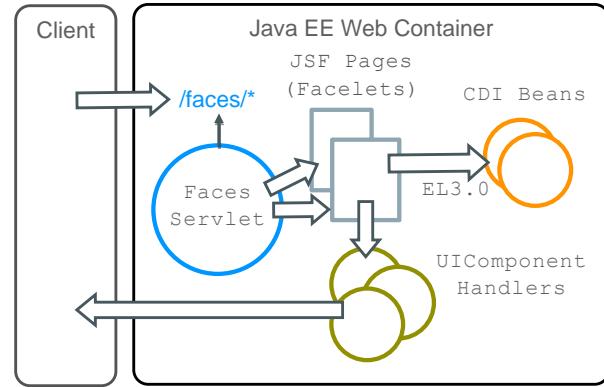
Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



JavaServer Faces Concepts

JavaServer Faces is a server-side component framework for building Java web applications. Its essential components are:

- Faces Servlet
 - Manages JSF state and lifecycle
 - Controls navigation between JSF pages
- JSF Pages (Facelets) are XHTML documents
 - Contain JSF UIComponents
 - Contain HTML5 elements
 - Other components defined by Libraries
- JSF UIComponent Handlers
 - Referenced from JSF pages Libraries
 - Produce output to the client
- CDI Beans
 - Handle business logic, maintain application state
 - Referenced from JSF pages using EL3.0 expressions
- Facelets may not directly contain scriptlets or Java code. Therefore, they represent pure View tier of the MVC, Faces Servlet acts as an MVC Controller and CDI Beans represent the Model.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

JavaServer Faces 2.2 standard is described by the JSR-344:

<https://www.jcp.org/en/jsr/detail?id=344>

JavaServer Faces technology consists of the following:

- An API for representing components and managing their state; handling events, server-side validation, and data conversion; defining page navigation; supporting internationalization and accessibility; and providing extensibility for all these features
- Tag libraries for adding components to web pages and for connecting components to server-side objects

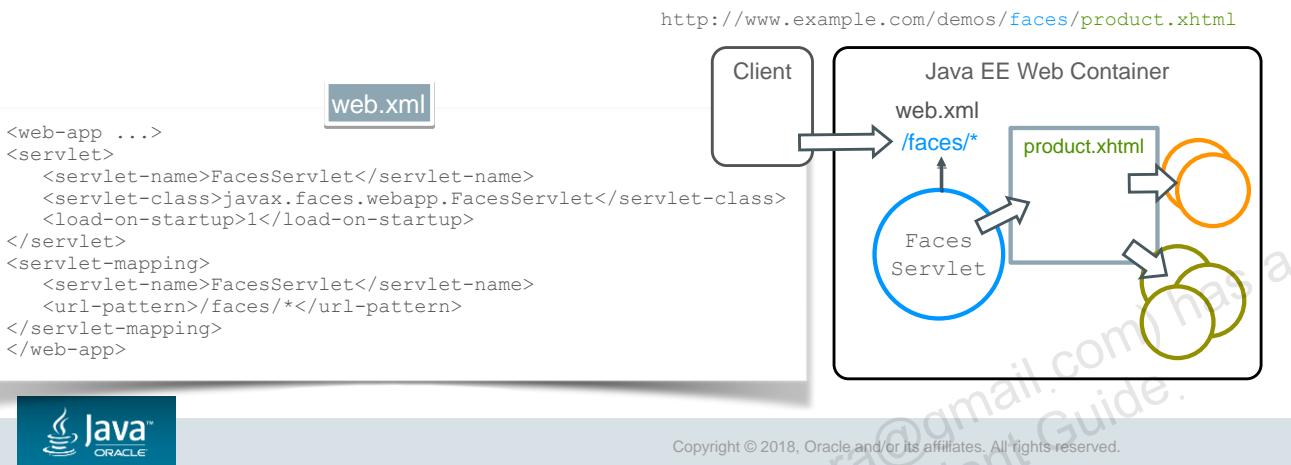
JSF page can be saved into a file that has any extension you like, typically jsf or xhtml

JSF UIComponent can produce and output itself, or with the help of the corresponding Renderer class. JSF implementations may often provide alternative Render Kits to control how component is displayed to the user, based on the selected rendererType property for corresponding UIComponent.

Faces Servlet Registration

Client invocation of a FacesServlet triggers the JSF request processing lifecycle

- FacesServlet is mapped to a url with web.xml file
- FacesServlet configuration may define tuning parameters such as state management, registry of additional tab libraries, etc...
- When client invokes a JSF page it actual calls upon FacesServlet to handle this page

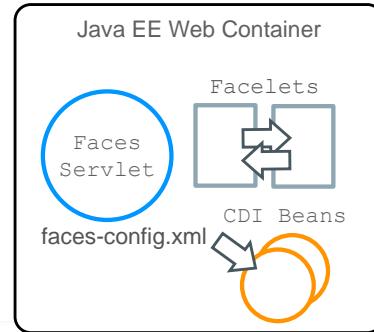


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

JSF Configuration

JSF runtime configuration is stored in the faces-config.xml file located in the WEB-INF folder

- An application configuration resource file may contain registration of managed beans, supported locales, navigation rules, validators, converters and references to other resources
- Unlike Managed Beans, CDI beans are configured via annotations and do not have to be declared in faces-config.xml.



faces-config.xml

```
<?xml version="1.0" encoding='UTF-8'?>
<faces-config version="2.2"
    xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
    http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig_2_2.xsd">
...
</faces-config>
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

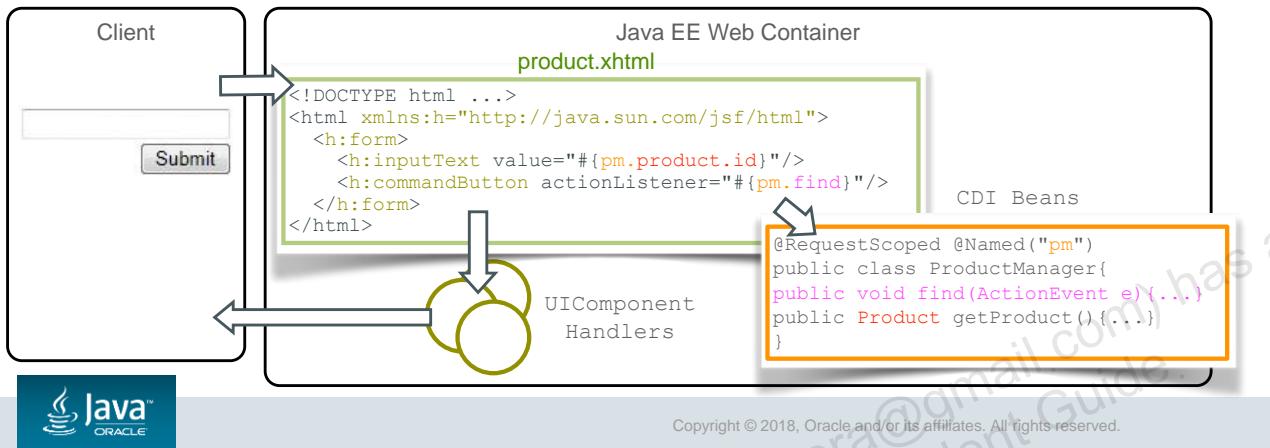
You can have more than one application configuration resource file for an application. The JavaServer Faces implementation finds the configuration file or files by looking for the following.

- A resource named /META-INF/faces-config.xml in any of the JAR files in the web application's /WEB-INF/lib/ directory and in parent class loaders. If a resource with this name exists, it is loaded as a configuration resource. This method is practical for a packaged library containing some components and renderers. In addition, any file with a name that ends in faces-config.xml is also considered a configuration resource and is loaded as such.
- A context initialization parameter, javax.faces.application.CONFIG_FILES, in your web deployment descriptor file that specifies one or more (comma-delimited) paths to multiple configuration files for your web application. This method is most often used for enterprise-scale applications that delegate to separate groups the responsibility for maintaining the file for each portion of a big application.
- A resource named faces-config.xml in the /WEB-INF/ directory of your application. Simple web applications make their configuration files available in this way.

JSF Facelet Structure

Facelets is a JSF standard declaration language for describing view structure of a JSF page

- Facelets may contain
 - HTML elements that are passed directly to the client
 - UIComponents described via Component Libraries and with the help of rendering kit generate output to the client
- CDI Beans handle business logic, provide values and are referenced with EL3.0 expressions



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Facelets are saved in XHTML files. This is an example of an XHTML declaration:

```
<?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>
...
</html>
```

CDI Beans can be defined using annotations

```
@RequestScoped("pm")
public class ProductManager {
    private int id;
    ...
}
```

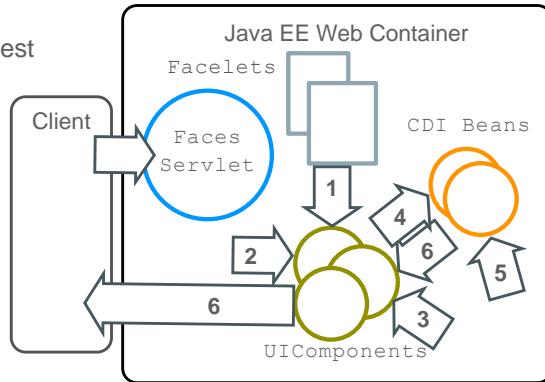
or using pre-CDI managed beans with faces-config.xml file

```
<managed-bean>
    <managed-bean-name>pm</managed-bean-name>
    <managed-bean-class>demos.ProductManager</managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
```

JSF Request-Response Lifecycle

JSF Application lifecycle consist of six phases:

- 1.Create/Restore View
 - Convert Facelet to the UIComponent tree on initial request
 - Restore UIComponent tree on postback requests
- 2.Apply Request Values
 - Update UIComponents with request data
 - Apply datatype conversions
- 3.Process Validations
 - Apply JSF and Bean Validation
- 4.Update Model Values
 - Copy UIComponent data to CDI beans (via EL)
- 5.Invoke Application
 - Execute action methods
- 6.Render Response
 - Read data from CDI beans (via EL)
 - Generate and send output. If navigation has occurred, the output would be produced by the target page



- ❖ In case of an error in any phase - Render Response phase is triggered
- ❖ On initial request only phases 1 and 6 are executed



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

You can also control your applications progression through the phases of the JSF lifecycle using FacesContext class. For example, you can interrupt JSF lifecycle and "skip" to render response phase using FacesContext.responseComplete() method.

The UIComponent tree is built using the structure contained in a Facelet page. Although an HTTP request triggers the creation of a UIComponent tree, the tree itself will outlive the initial request and response. JSF will use a render kit to convert the UIComponent tree to a suitable output format, such as an HTTP response containing an HTML body. The UIComponent tree is stored in FacesContext. Subsequent requests from a client will reuse the UIComponent tree.

Dynamic parts of the UIComponent tree are saved between requests. This state of the UIComponent tree is stored on the server by default. You can store the state on the client (in a hidden form field) to reduce server memory utilization and avoid view timeouts at the expense of additional bandwidth utilization. In web.xml, you can add:

```
<context-param>
  <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
  <param-value>client</param-value>
</context-param>
<context-param>
  <param-name>javax.faces.PARTIAL_STATE_SAVING</param-name>
  <param-value>false</param-value>
</context-param>
```

You can make a JSF page stateless by setting transient attribute to true

```
<html xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core" >
<f:view transient="true">
    rest of the page content
</f:view>
```

This is a new JSF 2.2 feature and is useful in cases where there is no need to maintain state, such as pages with read-only content. It helps the application to scale better.

Doing so will disable JSF ViewScope, so you would not be able to use ViewScoped beans.

JSF Libraries

JSF Libraries describe collections of components that are used to compose Facelets

| Tag Library | URI | Prefix | Purpose |
|-----------------------------------|---|--------|---------------------------------------|
| JavaServer Faces Facelets Library | http://xmlns.jcp.org/jsf/facelets | ui: | JSF tags for templating |
| JavaServer Faces HTML Library | http://xmlns.jcp.org/jsf/html | h: | JSF tags for all UIComponents |
| JavaServer Faces Core Library | http://xmlns.jcp.org/jsf/core | f: | JSF custom action tags |
| Pass-through Elements Library | http://xmlns.jcp.org/jsf | jsf: | Tags to support HTML5-friendly markup |
| Pass-through Attributes Library | http://xmlns.jcp.org/jsf/passthrough | p: | Tags to support HTML5-friendly markup |
| Composite Component Library | http://xmlns.jcp.org/jsf/composite | cc: | Tags to support composite components |
| JSTL Core Library | http://xmlns.jcp.org/jsp/jstl/core | c: | JSTL 1.2 Core Tags |
| JSTL Functions Library | http://xmlns.jcp.org/jsp/jstl/functions | fn: | JSTL 1.2 Functions Tags |

- Third-party libraries are also available to use with JSF. You can also create your own libraries.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

For more information see JSF 2.2 View Declaration Language reference:

<https://docs.oracle.com/javaee/7/javaserver-faces-2-2/vdldocs-facelets/toc.htm>

For more information on how to create custom UI and composite components see Java EE 7 tutorial sections 14 and 15:

<https://docs.oracle.com/javaee/7/tutorial/jsf-advanced-cc.htm#GKHXA>

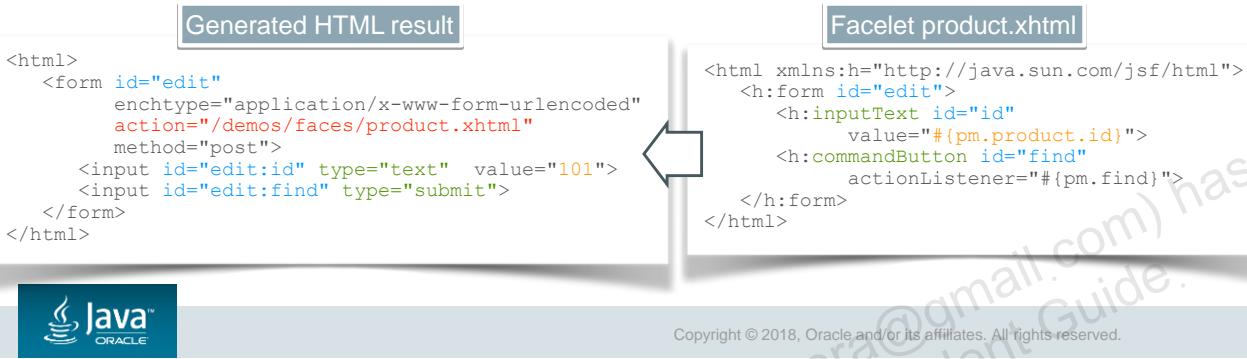
<https://docs.oracle.com/javaee/7/tutorial/jsf-custom.htm#BNAVG>

JavaServer Faces work with standard Java Server Pages tag libraries (JSTL)

JSF HTML Library UIComponents

JSF HTML Library describes UIComponents and HTML RenderKit

- UIComponents may hold application values
- Reference CDI beans with EL3.0 expressions
- Produce output to the client
- Dynamic information is handled with HTML forms
 - Forms generated by the h:form UIComponent always submit information back to the same page that has generated the form
 - FacesServlet intercepts all calls to facelets and handles submitted information as a part of JSF lifecycle



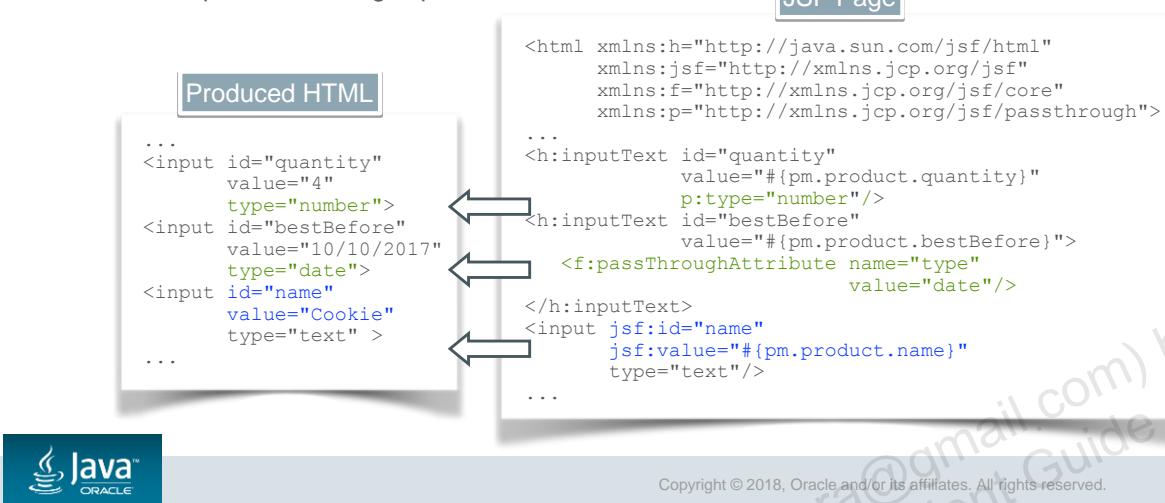
Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

JavaServer Faces specification implies that a UIComponents must produce HTML and may also produce other types of output. Different rendering kits can be created to produce alternative outputs.

JSF HTML Passthrough

JSF HTML Tag Library components are designed to produce HTML output to the client. However, you may also produce direct HTML.

- Add pass-through attributes to JSF UI Components
- Add JSF component handling capabilities to HTML elements



Elements specific to the client or markup are defined in renderer classes. To offer higher level of flexibility and to avoid rendering client-specific logic in JSF component classes the pass-through attributes are used. These attributes are going to be processed in the run time and at the client-side. Pass-through elements allow you to use HTML5 tags and attributes but to treat them as equivalent to JavaServer Faces components associated with a server-side UIComponent instance. The JSF implementation uses the element name and the identifying attribute to determine the corresponding Facelets tag that will be used in the server-side processing. The browser, however, interprets the markup that the page author has written.

Using Validators and Converters

JavaServer Faces Core Tag Library provides components that can convert and validate form data

- Converters are used to convert text data enter by user in a form into java objects used by CDI beans
 - f:converter - references java converter object
 - f:convertDateTime - converts between text values and date-time values
 - f:convertNumber - converts between text values and numeric values
- Validators are used to validate values submitted through forms
 - f:validateDoubleRange - double range
 - f:validateLength - minimum and maximum length
 - f:validateLongRange - long range
 - f:validateBean - bean validator
 - f:validateRegex - regular expression
 - f:validateRequired - mandatory
 - f:validator - custom

```
<h:inputText value="#{pm.product.bestBefore}">
    <f:convertDateTime pattern="dd-MM-yyyy" />
</h:inputText>
<h:inputText id="price" value="#{pm.product.price}">
    <f:convertNumber currencyCode="GBP" type="currency"/>
    <f:validateDoubleRange minimum="0.99" maximum="999.99"/>
</h:inputText>
<h:inputText id="quantity" value="#{pm.product.quantity}">
    <f:converter converterId="javax.faces.Integer" />
    <f:validateLongRange minimum="1" maximum="10"/>
</h:inputText>
```

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



For more information, see JavaServer Faces Core Tag Library documentation:
<https://docs.oracle.com/javaee/7/javaserver-faces-2-2/vdldocs-facelets/toc.htm>

JSF Templates

Facelet templating is a way of enabling consistent look and feel across multiple JSF pages. It also helps sharing common layout code.

- Create JSF template with common boilerplate html code
- Add ui:insert elements within the template
- Create JSF pages based on a template using ui:composition element
- Assign content to template with ui:define and ui:include elements

JSF Page

```
<ui:composition
    xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
    template="/layout.xhtml">
    <ui:define name="title">Products</ui:define>
    <ui:define name="content">
        <ui:include src="Products.jsf"/>
    </ui:define>
</ui:composition>
```

JSF Template layout.xhtml

```
<html
    xmlns:ui="http://xmlns.jcp.org/jsf/facelets">
<head>
<title>
    <ui:insert name="title"></ui:insert>
</title>
</head>
<body>
<div>
    <ui:insert name="content"/>
</div>
</body>
</html>
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

To achieve better code reusability and improve UI consistency, developers can use Resource Library Contracts.

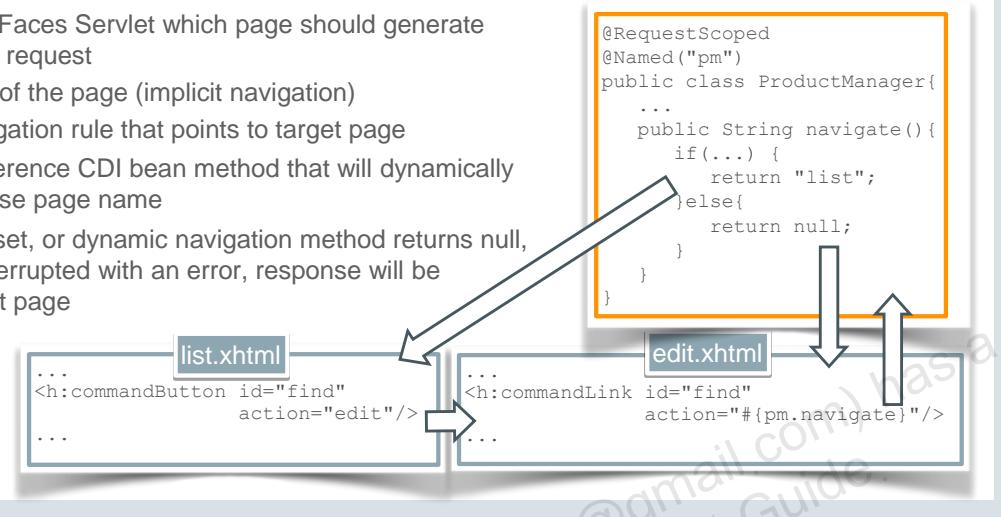
- Provide resources such as JSF templates, css files, images etc... placed into subfolders inside a WEB-INF\contracts folder (may be packed into a reusable jar file).
- Map each resource subfolder to a url using faces-config.xml file
- Switch contracts using <f:view contracts="contract-name"> attribute

For more information, see Applying JSF 2.2 Resource Library Contract tutorial: <http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/ResourceLibraryContract/resourceLibraryContract.html>

Describe JSF Navigation

Faces Servlet controls navigation between pages

- Each page always navigates recursively
- commandButton or commandLink UIComponents trigger page actions
- Action property informs Faces Servlet which page should generate response for the current request
 - This could be a name of the page (implicit navigation)
 - Or a name of the navigation rule that points to target page
- Action property may reference CDI bean method that will dynamically determine target response page name
- If action property is not set, or dynamic navigation method returns null, or lifecycle has been interrupted with an error, response will be generated by the current page



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Recursive nature of the JSF navigation means that a click on a commandLink or commandButton will submit the form contained within the JSF page back to the server. At this point Faces Servlet will start request-response handling lifecycle for the SAME page that has submitted the request (in a sense it feels like page has just submitted request to itself). Just before the point when the response will have to be rendered, Faces Servlet will see if navigation is required - i.e. if there is a not null action associated with this command link or a button. If there is such an action and no errors were encountered during the handling of this request, Faces Servlet will ask the target page (to which this action points) to produce the response, otherwise response is generated by the same page.

Configuring Navigation Rules

Navigation rules can also be set via faces-config.xml file

- **Navigation source** describes either a specific page or a wildcard
- **Navigation case outcome** must match action property set by navigation items such as commandButton or commandLink
- **Navigation target** defines a page that would be rendering response in a given navigation case
- Page may determine the navigation outcome dynamically by setting navigation action property to an expression that returns one of the outcomes

```
...
<navigation-rule>
    <from-view-id>/edit.xhtml</from-view-id>
    <navigation-case>
        <from-outcome>list</from-outcome>
        <to-view-id>/productList.xhtml</to-view-id>
    </navigation-case>
    <navigation-case>
        <from-outcome>view</from-outcome>
        <to-view-id>/productView.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
<navigation-rule>
    <from-view-id>*</from-view-id>
    <navigation-case>
        <from-outcome>home</from-outcome>
        <to-view-id>/home.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
...
```

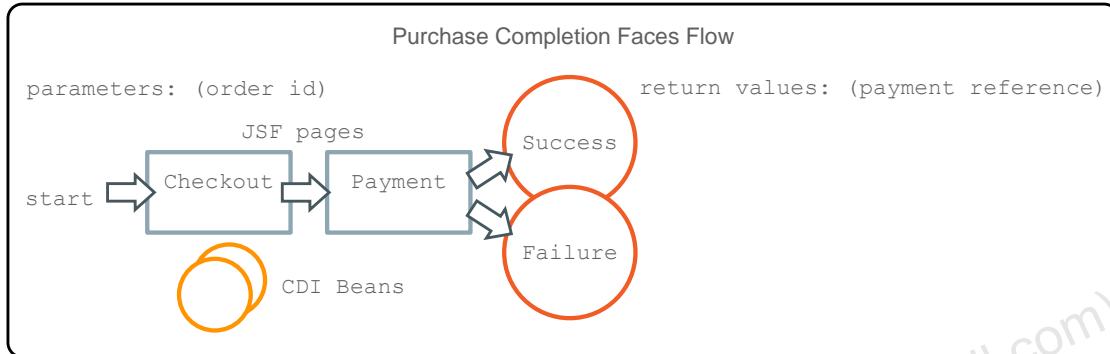
Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Using Faces Flows

Faces Flow feature allows you to create a set of pages, with its own scope:

- Has well defined entry point, may have parameters and return a values
- Has its own scope, @FlowScoped, that is similar to a Session, but is limited by flow boundaries
- Flows can be configured programmatically with @FlowDefinition annotation, or declaratively with face-config.xml, or separate flow configuration files per flow.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The Faces Flows feature of JavaServer Faces technology allows you to create a set of pages with a scope, FlowScoped, that is greater than request scope but less than session scope. For example, you might want to create a series of pages for the checkout process in an online store.

For more information on page flows is available in JSF tutorial:
<https://docs.oracle.com/javaee/7/tutorial/jsf-configure003.htm>

Action and ActionListener Attributes

JSF command components such as Link and Button may use Action and Action Listener attributes

- Action attribute may reference:
 - Target navigation page name
 - Navigation rules outcome name
 - An operation in a CDI bean that can execute business logic and return a name of a target page or navigation rule outcome
- ActionListener attribute may reference:
 - An operation in a CDI bean that can execute business logic
 - ActionListener handler is executed before the Action handler

```
<!DOCTYPE html ...>
<html xmlns:h="http://java.sun.com/jsf/html">
  <h:form>
    <h:commandButton
      actionListener="#{pm.doThings}"
      action="#{pm.doThingsAndNavigate}" />
  </h:form>
</html>
```

```
@RequestScoped @Named("pm")
public class ProductManager{
  public void doThings(ActionEvent e){...}
  public String doThingsAndNavigate(){...}
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Value and Binding Attributes

All JSF UIComponents may use Value and Binding attributes

- Value attribute references the value of the component
- Binding attribute references UIComponent object
- Referencing UIComponent object may be useful if you want to programmatically:
 - Alter component properties
 - Produce component specific messages
 - etc...

```
<!DOCTYPE html ...>
<html xmlns:h="http://java.sun.com/jsf/html">
  <h:form>
    <h:inputText
      value="#{pm.product.id}"
      binding="#{pp.pIdField}" />
  </h:form>
</html>
```

```
@RequestScoped @Named("pm")
public class ProductManager {
  private Product product;
  public Product getProduct() {...}
  public void setProduct(Product p) {...}
}
```

```
@RequestScoped @Named("pp")
public class ProductPageBackingBean {
  private HtmlInputText pIdField;
  public HtmlInputText getPIdField() {...}
  public void setPIdField(HtmlInputText pid) {...}
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Using immediate attribute

Immediate attribute allows component to be processed (applied, validated, updated etc..) immediately at **apply request values** phase of the JSF lifecycle

Immediate attribute can be applied to:

- Input components – to process these components ahead of others
- Command components – to perform actions without submitting the form
- Both – to partially submit form

```
<!DOCTYPE html ...>
<html xmlns:h="http://java.sun.com/jsf/html">
  <h:form>
    <h:inputText value="#{pm.product.id}" />
    <h:inputText value="#{pm.product.name}" />
    <h:inputText value="#{pm.product.price}" />
    <h:inputText value="#{pm.product.discount}" immediate="true" />
    <h:commandButton action="#{pm.applyDiscount}" immediate="true" />
  </h:form>
</html>
```

JSF Lifecycle Phases
(reminder)

1. Create/Restore View
- 2. Apply Request Values**
3. Process Validations
4. Update Model Values
5. Invoke Application
6. Render Response



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

There are three use-cases for the `immediate="true"` attribute:

- Setting immediate attribute for the input components such as input text would cause these items to process validations earlier - at apply request values phase. If validation or conversion fails for any such items other items in the page (non-immediate) would not be applied.
- Setting immediate attribute for the command components such as command button would skip the entire apply request values phase for any other input components. This can be used for disabling form fields validations for actions such as "cancel", "back" and sometimes "delete".
- Setting immediate attribute for both input and command components in the same form would cause this form to skip any non-immediate fields. This can be used to partially submit form values.

Using FacesContext Object

FacesContext object allows programmatic access to JSF runtime environment from CDI beans

- FacesContext object holds information about currently processed request
- FacesContext allows developers to:
 - Identify and control current JSF lifecycle phase
 - Retrieve and produce messages
 - Create and execute EL 3.0 expressions
 - etc...

```
@RequestScoped("sm")
public class SomeBean {
    public void someOperation() {
        FacesContext ctx = FacesContext.getCurrentInstance();
        ELContext elCtx = ctx.getELContext();
        ExpressionFactory ef = ctx.getApplication().getExpressionFactory();
        ValueExpression vex = ef.createValueExpression(elCtx, "#{pm.product}", Product.class);
        Product product = (Product)vex.getValue(elCtx);
    }
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

For more information see FacesContext object documentation:

<https://docs.oracle.com/javaee/7/api/javax/faces/context/FacesContext.html>

For more information on executing EL3.0 expressions see ELContext object documentation:

<https://docs.oracle.com/javaee/7/api/javax/el/ELContext.html> and ExpressionFactory documentation:

<https://docs.oracle.com/javaee/7/api/javax/el/ExpressionFactory.html>

JSF Localization

JSF configuration and libraries support localizations

- Localizable resources such as prompts, formats and messages are stored in resource bundles
- Each resource bundle should provide a translated version for every country and language
- Resource bundles are registered in the faces-config.xml file and are associated with variables that can be referenced in EL expressions
- JSF page may set preferred locale with a view component locale property



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Locale value can be a language code such as "en" or "fr", it can also distinguish countries by setting both language and country codes "en_US" or "en_GB"

Resource Bundle is used to define custom messages.

Message Bundle is used to override default JSF error and warning messages:

<message-bundle>demos.Messages</message-bundle>

Provide translated versions of this file:

demos.Messages_en.properties

javax.faces.component.UIInput.REQUIRED = {0}: This field is required

demos.Messages_ru.properties

javax.faces.component.UIInput.REQUIRED = {0}: Это обязательное поле

Consider providing translations for ValidationMessages.properties if your application is using BeanValidation constraints.

Displaying Messages

There are two UIComponents in the JavaServer Faces HTML Library that display messages

- Messages UIComponent displays messages for entire page
- Message UIComponent displays messages for a specific UIComponent within this page
- You can display different types of messages that represent errors, warnings or simple information messages
- Both may use direct css styles or reference different css style classes for different types of messages
- If there are no messages to display Message and Messages UIComponents produce no output
- To define custom error messages and translations for validators use ValidationMessages_xx_XX.properties files placed into WEB-INF/classes folder

ValidationMessages.properties

```
javax.validation.constraints.DoubleRange.message=Value must be between {min} and {max}
```

```
<h:messages fatalClass="fatalCSS"
    errorClass="errorCSS"
    warnClass="warnCSS"
    infoClass="inforCSS"/>
<h:inputText id="price"
    value="#{pm.product.price}">
    <f:validateDoubleRange minimum="0.99"
        maximum="999.99"/>
</h:inputText>
```

```
<h:inputText id="price"
    value="#{pm.product.price}">
    <f:validateDoubleRange minimum="0.99"
        maximum="999.99"/>
</h:inputText>
<h:message for="price"
    errorClass="errorCSS"/>
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Producing Messages From CDI Beans

CDI Beans may propagate messages to display on a page via the FacesContext object

- Publish message using addMessage method of the FacesContext object
- When creating a message you may set:
 - Severity Level
 - Message summary text
 - Message detail text

```
@RequestScoped @Named("pm")
public class ProductManager {
    public void updateProduct(Product p) {
        FacesContext ctx = FacesContext.getCurrentInstance();
        FacesMessage msg = null;
        try {
            ...
            msg = new FacesMessage(FacesMessage.Severity.SEVERITY_INFO,
                                   "Product updated successfully",null);
        }catch(Exception e) {
            msg = new FacesMessage(FacesMessage.Severity.SEVERITY_ERROR,
                                   "Error updating Product",e.getMessage());
        }
        ctx.addMessage(msg);
    }
}
```


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

To create a message associated with a specific UIComponent within the page, CDI bean has to:

- Reference target UIComponent
- Use this reference to find this components client id
- Associate this client id value with the message, when adding it to faces context object

```
@RequestScoped
@Named("pm")
public class ProductManager {
    private UIComponent priceItem;
    public void setPriceItem(UIComponent priceItem) {
        this.priceItem = priceItem;
    }
    public UIComponent getPriceItem() {
        return priceItem;
    }
}
```

```
public void updatePrice(float price) {  
    try{...}catch(Exception e) {  
        FacesMessage msg = new FacesMessage(  
            FacesMessage.Severity.SEVERITY_ERROR,  
            "Error updating product price",  
            ex.getMessage());  
        FacesContext ctx = FacesContext.getCurrentInstance();  
        ctx.addMessage(priceItem.getClientId(ctx), msg);  
    }  
}  
}  
}
```

Using Managed Properties

CDI Beans can be simply injected. Managed Beans may reference each other using managed properties

- ManagedProperty annotation or faces-config.xml file can be used to define managed bean reference
- Expressions are used to resolve bean references

```
<managed-bean>
    <managed-bean-name>om</managed-bean-name>
    <managed-bean-class>demos.OrderManager</managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
<managed-bean>
    <managed-bean-name>pm</managed-bean-name>
    <managed-bean-class>demos.ProductManager</managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
    <managed-property>
        <property-name>orders</property-name>
        <value>#{om}</value>
    </managed-property>
</managed-bean>
```

```
@RequestScoped
@Named("om")
public class OrderManager {
    ...
}
```

```
@RequestScoped("pm")
public class ProductManager {
    @ManagedProperty(value = "#{om}")
    private OrderManager orders;
    public void setOrders(OrderManager orders) {
        this.orders = orders;
    }
    ...
}
```

- ❖ You must make sure that when you set a managed bean as the property of another bean, their scopes are compatible. RequestScoped beans can reference SessionScoped beans, but not the other way around.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Note: You must make sure that when you set a managed bean as the property of another bean, their scopes are compatible. In general, the shorter-scoped beans can have the managed properties of the longer-scoped beans, but not the reverse. For example RequestScoped bean can access SessionScoped bean - such reference is not ambiguous.

Adding AJAX code to Facelets

Asynchronous JavaScript and XML (AJAX) is used when dynamic handling of values is required within a page without reloading the entire page

- Facelets can add build-in AJAX resource library support - jsf.js as well as any other JavaScript Libraries
- JavaScript handlers can be added to UIComponents to react to different JavaScript events
- AJAX component can refresh UIComponents within a page without reloading to the entire page
- JavaScript support allows JSF pages to utilise technologies such as REST Services or WebSockets

```
<h:form>
    <h:outputScript name="jsf.js" library="javax.faces" target="head"/>
    <h:inputText id="priceItem" value="#{pm.product.price}">
        <f:ajax event="change" execute="priceItem" render="discountItem"/>
    </h:inputText>
    <h:outputText id="discountItem" value="#{pm.product.discount != null ? pm.product.discount : 0}"/>
</h:form>
```

- ❖ For more information on AJAX refer to "JavaScript and HTML5: Develop Web Applications" training course



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

For more information on using AJAX with JavaServer Faces technology refer to Java EE 7 tutorial JSF-AJAX section: <https://docs.oracle.com/javaee/7/tutorial/jsf-ajax.htm>

Extended JSF Frameworks and Component Libraries

Additional Frameworks and Component Libraries for JavaServer Faces are available

- Produce sophisticated layouts, such as dynamic styling
- Add additional features and functionalities, such as complex Ajax support
- Increase developer productivity, such as utility code and automations
- JSF framework examples:
 - Apache MyFaces
 - Oracle ADF Faces
 - PrimeFaces
 - IceFaces
 - OmniFaces
 - And so on



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Summary

In this lesson you should have learned how to develop web applications using JavaServer Faces

- Describe JSF lifecycle and architecture
- Understand JSF syntax
- Use JSF Libraries
- Apply Validators and Converters to UIComponents
- Use UI templates
- Define navigation
- Handle localization
- Produce messages
- Use Expression Language (EL)
- Use CDI Beans
- Add AJAX support



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



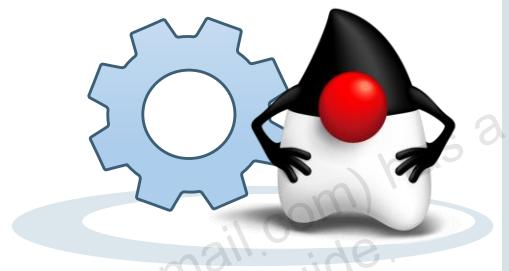
Practice

This practice covers the following tasks:

- Modify ProductManager to support JSF interactions
- Create JavaServer Faces application with pages to
 - Search for products
 - Display product list
 - Handle product update



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Jairo Jose Silvera Sarmiento (jairo.silvera@gmail.com) has a
non-transferable license to use this Student Guide.

Securing Java EE Applications



ORACLE®



12

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Jairo Jose Silvera Sarmiento (jairo.silvera@gmail.com) has a
non-transferable license to use this Student Guide.

Objectives

This chapter teaches how to use Java EE Security API to protect Applications

- Understand Java EE security architecture
- Configure Authentication using Login Modules
- Define Application Roles and Security Constraints
- Use programmatic security
- WebServices security standards



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



JAAS Security Concepts

Java Authentication and Authorization Services (**JAAS**) API helps developers to protect Java applications:

- **Authentication:** Establishing that the user's claimed identity is genuine
 - **Authorization:** Establishing that the user is allowed to carry out the requested action
- Java EE Servers can be configured to protect communications with transport level security by using a low-level, public-key protocol such as secure sockets layer (SSL):
- **Confidentiality:** Protecting data from unauthorized viewing during communication
 - **Integrity:** Ensuring that the data received is the same as the data sent



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Every application that is accessible over the web or in business-to-business environments must consider security. Your application must be protected from attack; the private data of your site's users must be kept confidential; and your services must also protect the clients and other servers.

Unfortunately, these considerations are never complete and security must always be viewed as a level of protection, not absolute protection. Recognizing this, system administration should include monitoring, so that when an attack occurs, it is noticed and can be stopped or the system disconnected before any major damage occurs.

Because of the sheer complexity, scope, and importance of the security problem, consider using specialist security experts in the creation of enterprise applications. However, each team member should also have as much understanding of major issues as is practical.

Authentication can be as simple as prompting for a username 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 containers and not by components.

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.

For confidentiality, the normal approach is to use encryption that is usually based on public-key techniques.

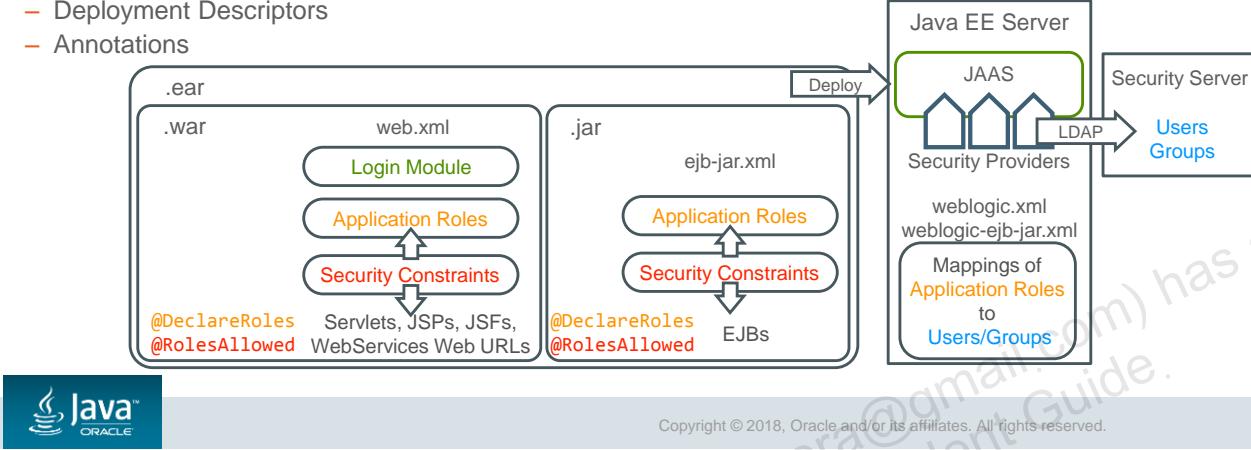
Integrity is normally achieved by signing the data. That is, the security model encrypts a checksum of the data and adds the encrypted checksum to the message to ensure that the message has not been modified during transmission.

SSL is now formally known as transport layer security (TLS). The use of TLS for component interactions is frequently standard in Java EE applications. Configuration of the transport is part of the administration of the application server and is not normally visible to the application developer.

JAAS Configuration

Java EE applications are secured with the help of the JAAS API

- JAAS provides portable security configuration with pluggable security providers
- Developers define Application Roles that are mapped to users and groups upon deployment
- Security Constraints restrict access to application components and are referencing Application Roles
- Login Modules provide a choice of authentication mechanisms
- Application security configurations can be provided with:
 - Deployment Descriptors
 - Annotations



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Java EE security contains no reference to the real security infrastructure, so that the application may be portable. JAAS (Java Authentication and Authorization Service) maps the application security policies to the security domain of the deployment environment. Thus, platform-independence is essentially preserved even though it becomes deployment-specific. JAAS implementations can provide for simple username-password pairs that are stored in a database, up to more complex systems such as Kerberos or Active Directory. In any case, the verification interface is standardized, although the means of adding new users is not. JAAS provides end-to-end security, so that security credentials are gathered in one part of the system, but are available to all parts. For example, if the user is using a web browser to interact with a servlet-based application and that servlet application calls enterprise beans, all the components (servlets and enterprise beans) that are invoked in a particular user interaction see the same user.

This portable configuration is achieved with following steps:

- Java EE server administrators configure security policy domains based on pluggable security service provider architecture, which enables various security providers, such as LDAP servers.
- Security Server administrators manage information about users and groups.
- Java application developers define Application Roles and describe security constraints that reference these Application Roles.
- Upon deployment Application Roles are mapped to actual users and groups defined by the security provider configured with security policy domain.
- Application developers select a Login Module, which defines a mechanism that allows to challenge the invoker to produce authentication.

To access an application, a user or an application may have to respond to the authentication challenge made by the Login Module.

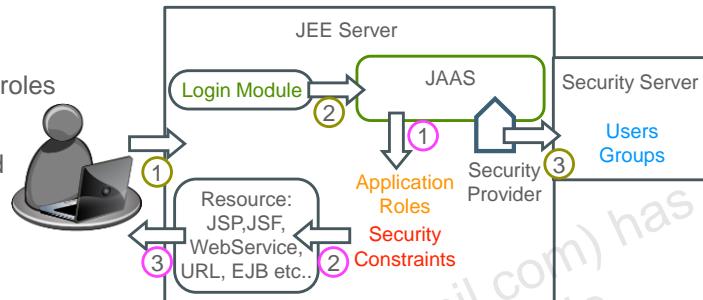
- Security Principal is a user or system that is authenticated with such society provider.
- Once Security Principal is established it can then be authorized to use application resources based on the mappings between this security principal and application roles that are referenced by application security constraints.
- Unauthenticated invoker may also be granted access to application resources, if security constraints allow that.

In addition to configuring authentication and authorization declaratively, developer can also use programmatic authorization controls provisioned by JAAS API.

Request Authentication and Authorization

A caller (user or another system) request authentication and authorization process

- Caller tries to invoke application resource, or explicitly access Login Module
- If a caller has not been authenticated yet and the invoked resource is protected by the security constant, then authentication process is triggered:
 1. Caller sends a request
 2. Login Module triggers JAAS authentication
 3. Security Provider validates credentials
- Then authorization is resolved:
 1. Security Principal is mapped to application roles
 2. Security Constraints are verified
 3. Access to the resource can now be granted



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Caller may access Login Module directly, or invocation of the Login Module can be triggered by the server when caller tries to access a resource protected with security constraints.

Unsuccessful authentication will prevent caller from accessing resources. Even if a caller has been successfully authenticated access to a resource can still be denied if the established Security Principal has no permissions to access resources protected by security constraints.

Login Module Configuration

Login Module describes authentication mechanism

- Java EE Server provide predefined Login Modules:
 - HTTP Basic Authentication
 - HTTP Digest Authentication
 - Form Authentication
 - Client Authentication
 - Mutual Authentication

web.xml

```
<login-config>
  <auth-method>FORM</auth-method>
  <realm-name>myrealm</realm-name>
  <form-login-config>
    <form-login-page>/login.html</form-login-page>
    <form-error-page>/error.html</form-error-page>
  </form-login-config>
</login-config>
```

login.html

```
<form method="POST" action="j_security_check">
  <input type="text" name="j_username">
  <input type="password" name="j_password">
  <input type="submit">
</form>
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

HTTP basic authentication requires that the server request a user name and password from the web client and verify that the user name and password are valid by comparing them against a configured security provider. Basic authentication is the default when you do not specify an authentication mechanism.

- A client requests access to a protected resource.
- The web server returns a dialog box that requests the user name and password.
- The client submits the user name and password to the server.
- The server authenticates the user in the specified realm and, if successful, returns the requested resource.

Like basic authentication, digest authentication authenticates a user based on a user name and a password. However, unlike basic authentication, digest authentication does not send user passwords over the network. Instead, the client sends a one-way cryptographic hash of the password and additional data. Although passwords are not sent on the wire, digest authentication requires that clear-text password equivalents be available to the authenticating container so that it can validate received authenticators by calculating the expected digest.

Form-based authentication allows the developer to control the look and feel of the login authentication screens by customizing the login screen and error pages that an HTTP browser presents to the end user. When form-based authentication is declared, the following actions occur.

- A client requests access to a protected resource.
- If the client is unauthenticated, the server redirects the client to a login page.
- The client submits the login form to the server.
- The server attempts to authenticate the user.
 - If authentication succeeds, the authenticated user's principal is checked to ensure that it is in a role that is authorized to access the resource. If the user is authorized, the server redirects the client to the resource by using the stored URL path.
 - If authentication fails, the client is forwarded or redirected to an error page.

With client authentication, the web server authenticates the client by using the client's public key certificate. Client authentication is a more secure method of authentication than either basic or form-based authentication. It uses HTTP over SSL (HTTPS), in which the server authenticates the client using the client's public key certificate. SSL technology provides data encryption, server authentication, message integrity, and optional client authentication for a TCP/IP connection. You can think of a public key certificate as the digital equivalent of a passport. The certificate is issued by a trusted organization, a certificate authority (CA), and provides identification for the bearer.

With mutual authentication, the server and the client authenticate each other. Mutual authentication is of two types:

- Certificate-based:
 - A client requests access to a protected resource.
 - The web server presents its certificate to the client.
 - The client verifies the server's certificate.
 - If successful, the client sends its certificate to the server.
 - The server verifies the client's credentials.
 - If successful, the server grants access to the protected resource requested by the client.
- User name/password-based:
 - A client requests access to a protected resource.
 - The web server presents its certificate to the client.
 - The client verifies the server's certificate.
 - If successful, the client sends its user name and password to the server.
 - The server verifies the client's credentials
 - If the verification is successful, the server grants access to the protected resource requested by the client.

Programmatic Authentication

Authentication can be performed programmatically with HttpServletRequest object methods:

- Method **authenticate** displays a login dialog box and collects the user name and password for authentication purposes.
- Method **login** allows an application to provide user name and password information collected in any custom way, other than specifying form-based authentication in an application deployment descriptor.
- Method **logout** allows an application to reset the caller identity of a request.

```
@WebServlet(name = "ProductDisplay", urlPatterns = {"/products"})
public class ProductDisplayServlet extends HttpServlet {
    protected void processRequest(HttpServletRequest request,
                                  HttpServletResponse response)
        throws ServletException, IOException {
        ...
        request.login(username,password);
        ...
        request.logout();
        ...
    }
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

An example of authenticate method of the HttpServletRequest object:

```
package demos;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class AuthTestServlet extends HttpServlet {
    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            request.authenticate(response);
            out.println("Authenticate Successful");
        } finally {
            out.close();
        }
    }
}
```

Declare Application Roles

Developers describe application security roles that are mapped to users and groups upon deployment

- Application Security Roles can be defined with
 - Standard Java EE deployment descriptors
 - DeclareRoles Annotation
- Server specific descriptors are used to map application roles to users and groups

Servlet or EJB

```
@DeclareRoles({"employee", "customer"})
```

web.xml or ejb-jar.xml

```
<security-role>
    <role-name>employee</role-name>
</security-role>
<security-role>
    <role-name>customer</role-name>
</security-role>
```

weblogic.xml or weblogic-ejb-jar.xml

```
<security-role-assignment>
    <role-name>employee</role-name>
    <principal-name>joe</principal-name>
    <principal-name>clerks</principal-name>
    <principal-name>managers</principal-name>
</security-role-assignment>
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Define Security Constraints

Security constraints restrict access to resources

- Web modules security constraints restrict access to web resources such as WebServices, Servlets etc..
- EJB modules security constraints restrict access to EJBs and their methods.



Java defines a number of security annotations:

- PermitAll - indicates that the given method or all business methods of an EJB are accessible by everyone
- DenyAll - indicates that the given method in the EJB cannot be accessed by anyone
- RolesAllowed - Indicates that the given method or all business methods in the EJB can be accessed by users associated with the list of roles
- DeclareRoles - Used by both Servlets and EJBs to define application security roles
- RunAs - Used by both Servlets and EJBs to specify the run-as role for the given components. By default components assume they are executed with the identity of the invoker

A web container managed resources can describe security constraints restarted to specific HTTP Methods. In the following example, only the POST method is restricted to the role customer:

```
<security-constraint>
    <display-name>Customer Product Permission</display-name>
    <web-resource-collection>
        <url-pattern>/product/*</url-pattern>
        <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>customer</role-name>
    </auth-constraint>
</security-constraint>
```

Or in this example annotations are configured to allow GET Method to be invoked by everyone and POST Method only by members of the employee role:

```
@WebServlet("/products")
@WebServlet(name = "ProductServlet", urlPatterns = {"/products"})
@ServletSecurity(
    httpMethodConstraints = {@HttpMethodConstraint("GET"),
        @HttpMethodConstraint(value = "POST", rolesAllowed = {"employee"})})
public class ProductServlet extends HttpServlet {...}
```

Java EE Programmatic Security

Java EE components can handle security programmatically

- Web container managed components can use methods of HttpServletRequest or JAX-RS WebServices SecurityContext objects:
 - isUserInRole
 - getUserPrincipal
- EJBs can use methods of EJBContext object:
 - isCallerInRole
 - getCallerPrincipal

```
request.isUserInRole("employee");
String username = request.getUserPrincipal().getName();
```

```
@Resource
private SessionContext ctx;

ctx.isCallerInRole("employee");
String username = ctx.getCallerPrincipal().getName();
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Generally it is preferable to utilize declarative security in Java EE applications. However, with declarative approach security constraint may be too coarse gained. In the EJB module you may wish to programmatically access information about caller identity to provide audit information, such as recording user audit data into the database. When declaring security constraints for the Web UI, you may use a programmatic approach to define security permission for a specific fragment of the page or JSF component.

Web Service Security

REST Services do not define any security model of their own.

- They rely on HTTP protocol to implement security.
- REST Services security is practically identical to HTTP Servlet security.
- Java EE Security Constraints can restrict URL access for specific HTTP methods

SOAP Services are protocol independent and define their own security standards:

- WS-Policy: Is the framework for adding assertions to web services
- WS-SecurityPolicy: Defines the policy assertions used by secure web services
- WS-Security: Includes extensions used in a SOAP message to create content that is digitally signed or confidential
- WS-Addressing: Adds addressing information to the SOAP message



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

WS-Security

WS-Security defines a transport protocol independent standard for authenticating and protecting SOAP messages

Selective encrypting and signing of message parts

- XML Encryption
- XML Signature

Management of security tokens:

- X.509 Certificates
- SAML Tokens
- Kerberos Tickets
- UserName Tokens



- ❖ WS-Security and other Web Services policies are supported by provider specific tools, such as Oracle Web Services Manager (OWSM) and are not part of a Java EE 7 standard JAX-WS implementation.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

OASIS group is a consortium that defines standards used by web services, including WS-Security specification. For more information see: <https://www.oasis-open.org>

W3C XML Signature specification <http://www.w3.org/Signature/>

W3C XML Encryption specification <http://www.w3.org/Encryption/2001/>

Summary

In this lesson you should have learned how to use Java EE Security API to protect Applications

- Understand Java EE security architecture
- Configure Authentication using Login Modules
- Define Application Roles and Security Constraints
- Use programmatic security
- WebServices security standards



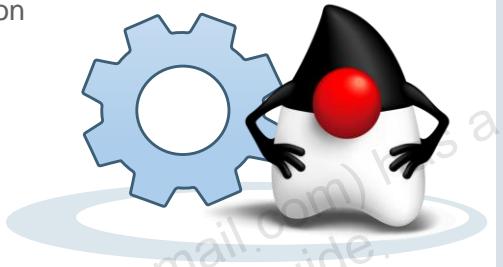
Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Practice

This practice covers the following tasks:

- Secure your web application by providing declarative security configuration
 - Configure Login Module
 - Add Application Security Roles
 - Define Security Constraints
 - Configure Security Provider
 - Map Applications Roles to Security Provider Groups
- Add programmatic security handling to your web application



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Java Logging



ORACLE®



A

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Jairo Jose Silvera Sarmiento (jairo.silvera@gmail.com) has a
non-transferable license to use this Student Guide.

Objectives

This appendix covers the following topics:

- Using Java Logging
- Configuring Logging Settings



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Java Logging Frameworks

Many logging frameworks exist that provide similar APIs:

- Logging libraries:
 - **log4j**: A popular logging framework from Apache
 - **java.util.logging (Java Logging)**: Built-in logging framework included since JDK 1.4
- Logging abstraction libraries:
 - Commons Logging: An abstraction of logging frameworks provided by Apache
 - Simple Logging Facade for Java (SLF4J): An alternative logging abstraction

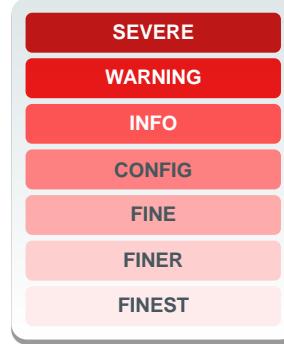


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Using Java Logging API

Use Logger class provided by Java Logging API to write logs.

- Each Logger is identified by a name.
- It is common practice to use class name as a logger name.
- There are seven levels of logging that allow you to log different levels of severity.



```
Logger logger = Logger.getLogger(demos.ProductManager.class.getName());
logger.log(Level.ERROR, "Your error message", throwable);
logger.log(Level.INFO, "Your message");
logger.info("Your message");
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

- The `java.util.logging.Level` class also defines `Level.ALL` and `Level.OFF`. These two levels are used for configuration and filtering purposes only.

Logging Method Categories

The logging methods are grouped into five main categories:

| Method | Purpose |
|--|---|
| log | This is the most commonly used logging method. These overloaded methods have parameters for a level, a message, and optional additional parameters. |
| logp | Similar to the log methods, the "log precise" methods take additional parameters that specify a class and method name. |
| logrb | Similar to the logp methods, the "log with resource bundle" methods take a resource bundle name that is used to localize the message. |
| entering exiting throwing | These are convenience methods used to log method entry, method exit, and exception throwing at the FINER level. |
| severe warning config info fine finer finest | These are convenience methods used in simple cases when a message should be logged at the level indicated by the method name. |



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Example of using different logger methods:

```
public static void main(String[] args) {
    logger.log(Level.INFO, "Application Starting");
    try {
        readFile();
    } catch (IOException e) {
        logger.log(Level.SEVERE, "Failed to read important file", e);
    }
}
```

```
public static void readFile() throws IOException {  
    logger.entering("JavaLogging", "readFile");  
    try {  
        Path path = Paths.get("doesnotexist.txt");  
        List<String> lines =  
            Files.readAllLines(path, Charset.defaultCharset());  
    } catch (IOException e) {  
        logger.throwing("JavaLogging", "readFile", e);  
        throw e;  
    } finally {  
        logger.exiting("JavaLogging", "readFile");  
    }  
}
```

Guarded Logging

Use guarded logging to avoid processing messages that are due to be discarded.

1. Logging level can be set programmatically via the configuration.
2. Message is concatenated, but is not recorded because it is below the logging level threshold.
3. Message is not processed if it is below the logging level threshold.
4. Object parameters can be used to avoid concatenating messages unnecessarily.

```
1 logger.setLevel(Level.INFO);
2 logger.log(Level.FINE, "Product "+id+" has been selected");
3 if(logger.isLoggable(Level.FINER)) {
4     logger.log(Level.FINE, "Product "+id+" has been selected");
}
logger.log(Level.FINE, "Product {0} has been selected", id);
```

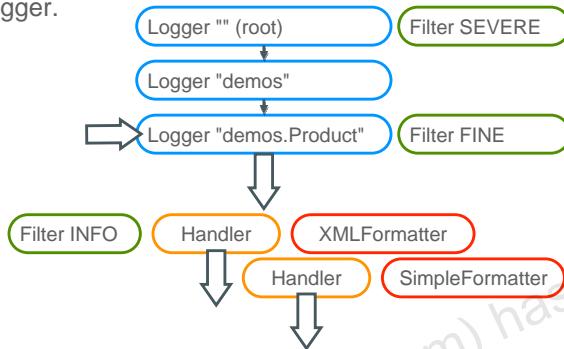


Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Log Writing Handling

Log records can be discarded (filtered out) by one or more filter that may be attached to a logger of a log handler.

- Logger writes log messages with different log levels.
- Loggers form a hierarchy.
 - Child Logger inherits log level from parent Logger.
 - Child Logger can override the log level.
- Log Handler writes log messages to a log destination:
 - Console
 - File
 - Memory
 - Socket
 - Stream
- Filters can be set for both Loggers and Log Handlers.
- Handlers use Formatters to format the record.
 - SimpleFormatter
 - XMLFormatter



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Because the name of a Logger defines its place in a hierarchy, the names you choose to give your loggers are important. The two most common names used are:

- The fully qualified classname of the class using logging statements
- The package name (not including the classname) of the class using logging statements

All Loggers are registered with `LogManager`. You can use the `java.util.logging.LogManager` class to list all the registered logger names.

```

LogManager lm = LogManager.getLogManager();
Enumeration<String> nameEnum = lm.getLoggerNames();
while(nameEnum.hasMoreElements()) {
    String loggerName = nameEnum.nextElement();
    Logger lgr = Logger.getLogger(loggerName);
}
  
```

In addition to existing log value filters, custom filters can also be created.

Example of the custom log filter:

```
package demos;  
import java.util.logging.Filter;  
public class CustomFilter implements Filter {  
    public boolean isLoggable(LogRecord record) {  
        // analyse log record and return true if record should be logged or  
        false if it should be discarded  
        return false;  
    }  
}
```

Logging Configuration

Logging can be configured using `logging.properties`:

```
handlers=java.util.logging.ConsoleHandler
demos.handlers=java.util.logging.FileHandler

.level=INFO
demos.level=FINE

java.util.logging.ConsoleHandler.formatter=java.util.logging.SimpleFormatter

java.util.logging.FileHandler.pattern=%h/java%u.log
java.util.logging.FileHandler.limit=50000
java.util.logging.FileHandler.count=1
java.util.logging.FileHandler.formatter=java.util.logging.XMLFormatter
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

By default, the `logging.properties` file located in `JAVA_HOME/jre/lib` is used to configure Java Logging. WebLogic Server can be either Java Logging (the default) or Log4J Logging.

The following example shows passing the `logging.properties` file in the -
`Djava.util.logging.config.file` argument to the `weblogic.Server` startup command:

```
java -Djava.util.logging.config.file=C:\mydomain\logging.properties
weblogic.Server
```

Programmers can also update the logging configuration programmatically at run time in several ways:

- FileHandlers, MemoryHandlers, and PrintHandlers can all be created with various attributes.
- New Handlers can be added and old ones can be removed.
- New loggers can be created and can be supplied with specific handlers.
- Levels can be set on target Handlers.

Application Server Logging Configuration

Application servers have their own logging features and configurations

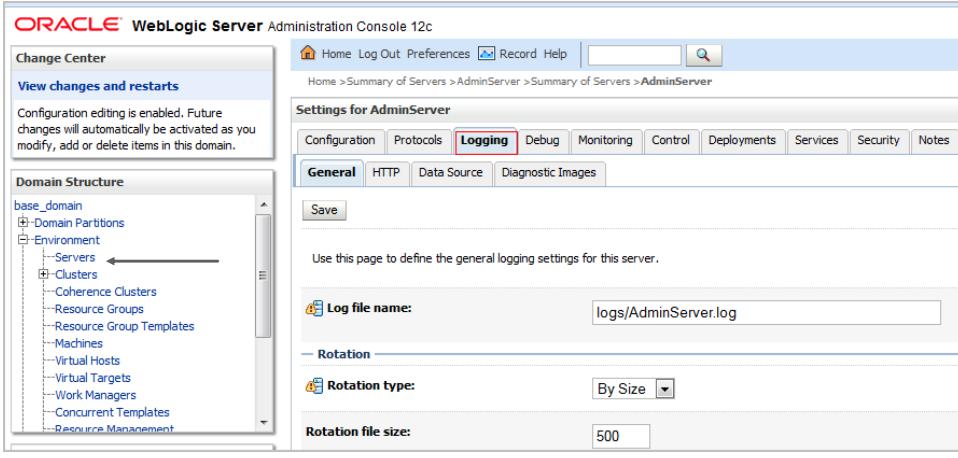
For example WebLogic server:

- Uses `java.util.logging`
- Writes log messages to:
 - The console
 - A server log file:
`<domain folder>/servers/<server folder>/logs/<server name>.log`
- Rotates the `server.log` file when it reaches specified size
- Publishes Server log entries via a REST interface
- Uses Oracle Diagnostics Logging (ODL) format



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Configuring the WebLogic Logging Service



The screenshot shows the Oracle WebLogic Server Administration Console 12c interface. On the left, there's a navigation pane titled "Change Center" with "View changes and restarts" and a note about configuration editing. Below it is the "Domain Structure" tree, which includes "base_domain", "Domain Partitions", "Environment" (selected), "Servers", "Clusters", "Coherence Clusters", "Resource Groups", "Resource Group Templates", "Machines", "Virtual Hosts", "Virtual Targets", "Work Managers", "Concurrent Templates", and "Resource Management". The main panel is titled "Settings for AdminServer" under the "Logging" tab. It has tabs for Configuration, Protocols, Logging (selected), Debug, Monitoring, Control, Deployments, Services, Security, and Notes. Under the Logging tab, there are sections for General, HTTP, Data Source, and Diagnostic Images. A "Save" button is present. The "General" section contains fields for "Log file name" (set to "logs/AdminServer.log"), "Rotation type" (set to "By Size"), and "Rotation file size" (set to "500"). A watermark across the page reads "Jairo Silveira (jairo.silveira@gmail.com) has a non-transferable license to use this Student Guide."

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

To configure logs:

1. In the left pane of the Console, expand Environment and select Servers.
2. In the Servers table, click the name of the server instance whose logging you want to configure.
3. Select Logging > General.
4. Retain or modify the default values.
5. Click Save.

Viewing WebLogic Server Logs

The screenshot shows the Oracle WebLogic Server Administration Console 12c interface. The left pane contains a 'Change Center' section with a message about configuration editing and a 'Domain Structure' tree. The 'Diagnostics' node is expanded, and the 'Log Files' node is highlighted with a red arrow. The right pane is titled 'Summary of Log Files' and contains a table of log files. The 'DomainLog' entry is selected, indicated by a red arrow. The table has columns for Name, Type, and Server. The 'Name' column shows 'DataSourceLog', 'DomainLog', and 'EventsDataArchive'. The 'Type' column shows 'Data Source Profile Log', 'Domain Log', and 'Instrumentation'. The 'Server' column shows 'AdminServer' for all three entries. A 'View' button is highlighted with a red box. At the bottom of the right pane, there is a copyright notice: 'Copyright © 2018, Oracle and/or its affiliates. All rights reserved.'

| Name | Type | Server |
|-------------------|-------------------------|-------------|
| DataSourceLog | Data Source Profile Log | AdminServer |
| DomainLog | Domain Log | AdminServer |
| EventsDataArchive | Instrumentation | AdminServer |

To view logs:

1. In the left pane of the Console, expand Diagnostics and select Log Files.
2. In the Log Files table, click the option button next to the name of the log you want to view.
3. Click View.
4. The page displays the latest contents of the log file; up to 500 messages in reverse chronological order. The messages at the top of the window are the most recent messages that the server has generated. The log viewer does not display messages that have been rotated into archive log files.
5. Click the option button next to the log record you want to view.
6. Click View. The page displays the log file entry.

Jairo Jose Silvera Sarmiento (jairo.silvera@gmail.com) has a
non-transferable license to use this Student Guide.

CDI Beans



ORACLE®



B

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Jairo Jose Silvera Sarmiento (jairo.silvera@gmail.com) has a
non-transferable license to use this Student Guide.

Objectives

This appendix covers the following topics:

- Defining Bean Qualifiers
- Using beans.xml
- Creating CDI Producers and Disposers
- Defining Interceptors
- Defining Event
- Defining Stereotypes



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Using Named Qualifiers

CDI Bean Qualifiers are used to reference a specific bean when:

- There is an ambiguity over which bean to inject
- @Named qualifier is also used by JSP/JSF pages

```
@Named("oo")
@RequestScoped
public class OnlineOrder implements Order { ... }
```

```
@WebServlet
public class OrderServlet extends HttpServlet {
    @Inject
    private @Named("oo") Order order;
}
```

```
<!DOCTYPE html ...>
<html xmlns:h="http://java.sun.com/jsf/html">
    <h:form>
        <h:inputText value="#{oo.id}">
    </h:form>
</html>
```

```
@Named("so")
@RequestScoped
public class StoreOrder implements Order { ... }
```

```
public class OrderApplication {
    @Inject
    private @Named("so") Order order;
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

There are two built-in qualifiers like @Named that are hidden by default.

- @Any: This qualifier is assigned to every bean. The @Any qualifier allows an injection point to refer to all beans or all events of a certain bean type.
- @Default: This is assigned to any bean that does not have a qualifier other than @Named. If you specify a qualifier that you define, it replaces @Default.

Every bean has the built-in qualifier, @Any. When a bean does not declare a qualifier other than @Named, the bean gets one additional qualifier of type @Default.

For example:

```
@Named
@RequestScoped
public class MessageOne implements Message{} Is equivalent to:
@Named
@RequestScoped
@Any
@Default
public class OnlineOrder implements Order{}
```

Using Custom Qualifiers

Custom qualifier annotation can also be used to inject a specific bean.

- You can still use @Named annotation for JSP/JSF pages.

```
@Qualifier  
@Retention(RUNTIME)  
@Target({METHOD, FIELD, PARAMETER, TYPE})  
public @interface Online { }
```

```
@Online  
@RequestScoped  
public class OnlineOrder implements Order {...}
```

```
@WebServlet  
public class OrderServlet extends HttpServlet {  
    @Inject  
    private @Online Order order;  
}
```

```
@RequestScoped  
public class StoreOrder implements Order {...}
```

```
public class OrderApplication {  
    @Inject  
    private Order order;  
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

When applied to a bean, the qualifier limits where the bean can be injected.

@Retention: How long this annotation should be retained during its use: SOURCE, CLASS, RUNTIME

Using Alternative Qualifiers

Alternative qualifier annotation can also be used to inject a specific bean.

- Injected type can be changed via the beans.xml file.

```
@Alternative  
@RequestScoped  
public class OnlineOrder implements Order {...}  
  
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"  
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee  
                           http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"  
       version="1.1" bean-discovery-mode="all">  
    <alternatives>  
        <class>demos.OnlineOrder</class>  
    </alternatives>  
</beans>  
  
@WebServlet  
public class OrderServlet extends HttpServlet {  
    @Inject  
    private Order order;  
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Producer and Disposer Methods

A CDI producer method generates bean instances that can be injected.

- Injects an object that is not itself a bean
- Allows complex object initialization

A disposer is invoked when the CDI context ends.

```
@RequestScoped
public class OnlineOrder implements Order { ... }
```

```
@RequestScoped
public class StoreOrder implements Order { ... }
```

```
public enum OrderType {
    ONLINE, STORE;
}
```

```
@Qualifier
@Retention(RUNTIME)
@Target({METHOD, FIELD, PARAMETER, TYPE})
public @interface OrderTypeQualifier {
    OrderType value();
}
```

```
@RequestScoped
public class OrderFactory {
    @Produces
    @OrderTypeQualifier(OrderType.ONLINE)
    public Order getOnlineOrder() {
        return new OnlineOrder(...);
    }
    @Produces
    @OrderTypeQualifier(OrderType.STORE)
    public Order getStoreOrder() {
        return new StoreOrder(...);
    }
    public void disposeOrder(@Disposes
        @OrderTypeQualifier Order order) {
        order.close();
    }
}
```

```
@WebServlet
public class OrderServlet extends HttpServlet {
    @Inject OrderTypeQuaifier(OrderType.ONLINE)
    private Order order;
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Producers help you to use classes that do not conform to CDI bean standard (for example, a class that has a constructor with parameters) as a CDI bean.

Disposers help you to perform a cleanup (if required by the bean) when the CDI context in which this bean is used ends.

Interceptors

Interceptors are used to execute code during the selected life-cycle events of a bean but do not affect business logic.

- Are good for logging or auditing tasks
- Allow these tasks to be defined in one place

The life-cycle events that can be intercepted include:

- `@AroundInvoke` (most common example)
- `@PostConstruct`
- `@PreDestroy`
- `@PrePassivate`
- `@PostActivate`

```
@RequestScoped
public class OnlineOrder{
    @Logging
    public void addProduct(Product p){...}
}
```

```
@Inherited
@InterceptorBinding
@Retention(RUNTIME)
@Target({METHOD,TYPE})
public @interface Logging {}

@Logging
@Interceptor
@Priority(Interceptor.Priority.APPLICATION)
public class OrderInterceptor {
    @AroundInvoke
    public Object writeLog(InvocationContext ctx)
        throws Exception {
        logger.log(...);
        Object obj = ctx.proceed();
        logger.log(...);
        return obj;
    }
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

By default, only classes that are included in a bean archive can be identified as interceptors. To change the scope of the interceptor, use the `@Priority` annotation.

For example:

```
@Priority(Interceptor.Priority.APPLICATION)
```

Interceptors can be registered in a beans.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                           http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
       version="1.1" bean-discovery-mode="all">
    <interceptors>
        <class>com.example.webapp.GreetingInterceptor</class>
    </interceptors>
</beans>
```

CDI Events

CDI events allow beans to communicate without any compile time dependencies.

- Beans can define an event.
- Any bean can fire an event.
- A bean event handler takes care of fired events.
- Events may be used in conjunction with qualifiers.
- Events are used with the `@Observes` annotation.

```
@RequestScoped  
public class Order {  
    @Inject Event<Product> addEvent;  
    public void addProduct(Product p) {  
        ...  
        addEvent.fire(p);  
    }  
}
```

```
@RequestScoped  
public class StockManagement {  
    public void reserveProduct(@Observes Product p) {...}  
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Stereotypes

A stereotype is an annotation that incorporates other annotations.

- Stereotypes can be particularly useful in large applications where you have a number of beans that perform similar functions.
- A stereotype may specify:
 - A default scope
 - Zero or more interceptor bindings
 - Optionally, a @Named annotation, guaranteeing default EL naming
 - Optionally, an @Alternative annotation, specifying that all beans with this stereotype are alternatives
- A bean annotated with a particular stereotype will always use the specified annotations, so you do not have to apply the same annotations to many beans.

```
@Alternative  
@Stereotype  
@Target(TYPE)  
@Retention(RUNTIME)  
@RequestScoped  
public @interface Custom {}
```

```
@Custom  
public class Order {}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Jairo Jose Silvera Sarmiento (jairo.silvera@gmail.com) has a
non-transferable license to use this Student Guide.

BeanValidation and JPA API



ORACLE®



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Jairo Jose Silvera Sarmiento (jairo.silvera@gmail.com) has a
non-transferable license to use this Student Guide.

Objectives

This appendix covers the following topics:

- Creating Custom BeanValidation Constraints
- Handling Entity Relationship mappings
- Mapping Composite Primary Keys
- Mapping Embeddable classes
- Mapping an Entity to multiple tables



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Custom BeanValidation Constraints

You can build custom constraints based on existing BeanValidation Constraints.

```
import static ElementType.*;
@Id
@NotNull
@Pattern.List({
    @Pattern(regexp="[A-Z]{3}-[0-9]{2}")
})
@Constraint(validatedBy = {})
@Documented
@Target({METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
public @interface ProductCode {
    String message() default "{invalid.product_code}";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
    @Target({METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER})
    @Retention(RetentionPolicy.RUNTIME)
    @Documented
    @interface List {
        ProductCode[] value();
    }
}
```

```
@Entity
public class Product {
    @ProductCode
    private String code;
    ...
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The `validatedBy` attribute of the Constraint annotation can be used to specify customer validation class, if programmatic constant validation is required.

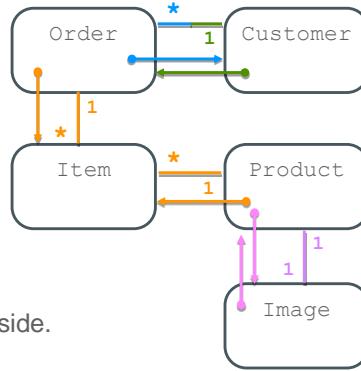
Entity Relationship Types

Entities can reference each other with different relationships:

- OneToOne
- ManyToOne
- OneToMany
- ManyToMany

Relationships can be:

- Bidirectional
 - Many side is an owning side
 - One side is an inverse side
 - In a OneToOne or ManyToMany case, you can choose the owning side.
- Unidirectional
 - Has only one side (owning)



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Ownership specifies the owning side of the relationship. In a unidirectional relationship, ownership is implied. The following terminology is used when discussing ownership:

- **Owning side:** In the Java Persistence API, the owning side determines which entity causes updates to the relationship in the database. The owning side is also called the *source side*.
- **Inverse side:** The non-owning side—or inverse side—of the relationship receives updates from the owning side. The non-owning side is also called the *target side*.

Direction implies navigation or visibility. Direction is expressed using one of the following terms:

- **Bidirectional:** In a bidirectional relationship, each entity can see the other entity in the relationship. You can include code in either entity that navigates to the other entity to obtain information or services from the other entity. In the Java Persistence API, the bidirectional nature of the relationship is specified by the use of one of the cardinality annotations in the entity classes on both sides of the relationship.
- **Unidirectional:** In a unidirectional relationship, only one of the entities can see the other entity in the relationship. In the Java Persistence API, the unidirectional nature of the relationship is specified by the use of one of the cardinality annotations in the entity class that owns the relationship.

The owning side of a relationship determines the updates to the relationship in the database.

Changes made to only the inverse side of an association are ignored.

The entity manager looks only at the owning side when determining the state of an association and, therefore, determines whether or not there is anything to do to update the association in the database.

Order entity can be designed to point to a Customer object - this would form a ManyToOne relationship.

If Customer entity is also designed to have way of printing to a Collection of Order objects via OneToMany relationship, this would make the relationship between the Order and Customer entities bidirectional.

If you model Item as a separate entity it could participate in two relationships - with Order and Product. However, you may choose not to create an entity for the item, but use it as a JoinTable to create a ManyToMany relationship between Order and Product.

A relationship between Product and Image is OneToOne, any one of its sides can be the owning side.

Mapping Entity Relationships

```

@Entity
public class Order {
    @Id
    private int id;
    @ManyToOne
    @JoinColumn(name="cus_id")
    private Customer customer;
    @ManyToMany
    @JoinTable(name="ITEM",
               joinColumns=@JoinColumns(name="ord_id",
                                         inverseJoinColumns=@JoinColumns(name="pro_id")))
    private Set<Product> items;
    ...
}

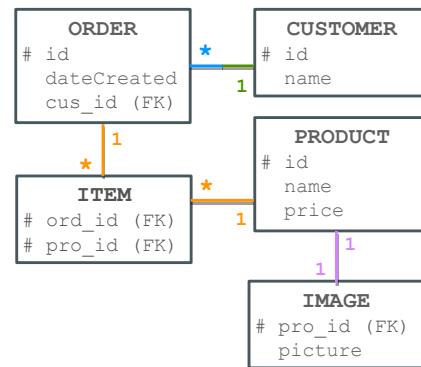
@Entity
public class Customer {
    @Id
    private int id;
    @OneToMany(mappedBy="customer")
    private Collection<Order> orders;
    ...
}

```

```

@Entity
public class Image {
    @Id
    @OneToOne(mappedBy="image")
    private Product product;
    ...
}

```



```

@Entity
public class Product {
    @Id
    private int id;
    @ManyToMany(mappedBy="items")
    private Set<Order> orders;
    @OneToOne
    @JoinColumn(name="id")
    private Image image;
    ...
}

```

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Owning side of the relationship is mapped with `JoinColumn` or `JoinTable` annotations pointing to database columns that implement your foreign keys.

Inverse side of the relationship mapping is using a `mappedBy` attribute to point to an owning side of the relationship.

Many side of the relationship is represented with a List, Collection, or Set.

Usually the owning side of the relationship is placed into the entity that represents the table that has got a the foreign key column and the inverse side is reflected in the entity that has the primary key to which this foreign key is referring. However, in a OneToOne relationship you may choose an opposite direction of the mapping, since foreign and primary keys are essentially reversible in this situation.

Entity Relationship Mapping Properties

Entity Relationship mapping properties are used to control the following:

- The order of elements in the list
- When removing child from collection and save parent, child is not left in the database as orphan, but is deleted from the database
- Cascade the removal of child elements when parent is removed
- EAGER or LAZY fetching
- Immediate EAGER fetch is the default for fetching Entity Attributes and ManyToOne relationships.
- Deferred LAZY fetch is the default for fetching Collections, OneToOne, OneToMany and ManyToMany relationships.

```
@Entity
public class Customer {
    @Id
    private int id;
    @OneToMany(mappedBy="customer", orphanRemoval=true, cascade=REMOVE, fetch=FetchType.EAGER)
    @OrderBy("dateCreated")
    private List<Order> orders;
    ...
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The OrderBy annotation can be applied to `java.util.List` collections, and the element selected must be comparable.

The default order is ascending if the DESC or ASC is not specified.

Multiple OrderBy attributes can be defined as a comma-separated list, as in this example:

```
@OrderBy("orderDate DESC, id ASC")
```

`cascade=REMOVE` means that when you remove parent entity object all of its children are also deleted

(only applicable to OneToOne or OneToMany relationships)

`orphanRemoval=true` means that you remove a child object from the collection that belongs to a particular parent object (in memory operation), and then merge this parent object (saving it into the database). This removed child record should be deleted from the underlying database.

Mapping Embeddable Classes

Embeddable classes represent a way to reuse commonly occurring groups of attributes.

- Similar to any other Entity class: Can have same mappings and even relationships of their own just like any other Entity
- Are embedded into other entities

```
@Entity
public class Person {
    @Id
    private int id;
    @Column(name="first_name")
    private String firstName;
    @Column(name="last_name")
    private String lastName;
    @Embedded
    private Address address;
    ...
}
```

```
@Entity
public class Company {
    @Id
    private int id;
    private String name;
    @Embedded
    private Address address;
    ...
}
```

```
PERSON
# id
first_name
last_name
street
city
post_code
...
```

```
COMPANY
# id
name
street
city
post_code
...
```

```
@Embeddable
public class Address {
    private String street;
    private String city;
    @Column(name="post_code")
    private String postCode;
    ...
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

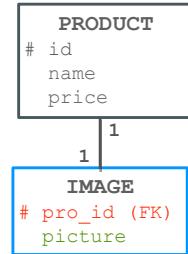
Mapping an Entity to Multiple Tables

Entity can be mapped to more than one table using SecondaryTable annotation.

You must specify:

- A Join condition
- An origin table for each secondary table attribute

```
@Entity
@Table(name="PRODUCT")
@SecondaryTable(name="IMAGE",
    pkJoinColumns=@PrimaryKeyJoinColumn(name="PRO_ID"))
public class Product {
    @Id
    private int id;
    private String name;
    private float price;
    @Lob
    @Basic(fetch=LAZY)
    @Column(table="IMAGE")
    private byte[] picture;
    ...
}
```



This examples also demonstrates mapping of a BLOB column.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Default fetch type for an attribute is EAGER. However, for performance reason you may override this using Basic annotation and set fetch type as LAZY. The example in the slide above uses LAZY annotation to disable default fetching of the picture when fetching product. Picture will only be fetched when actually needed.

When mapping a number of attributes from the Secondary table, consider using Embedded type with AttributeOverrides annotation:

```
@Entity  
@Table(name="PRODUCT")  
@SecondaryTable(name="IMAGE",  
    pkJoinColumns=@PrimaryKeyJoinColumn(name="PRO_ID"))  
public class Employee {  
    @Id private int id;  
    private String name;  
    private float price;  
    @Embedded  
    @AttributeOverrides ({  
        @AttributeOverride(name="picture",  
column=@Column(table="IMAGE")),  
        @AttributeOverride(name="size", column=@Column(table="IMAGE")),  
        @AttributeOverride(name="file_type",  
column=@Column(table="IMAGE"))  
    })  
    private Image image;  
    ...  
}
```

Composite Primary Key

Map multicolumn primary key using JPA API.

- Create a class that represents primary key components.
- Override equals and hashCode methods to compare primary key values.
- Use one of these approaches:

EmbeddedId

```
@Embeddable
public class ItemId
    implements Serializable {
    private Integer oid;
    private Integer line;
    ...
}

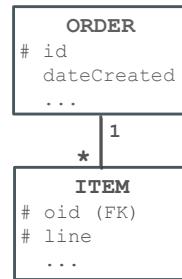
@Entity
public class Item {
    @EmbeddedId
    private ItemId id;
    ...
}
```

or

IdClass

```
public class ItemId
    implements Serializable {
    private Integer oid;
    private Integer line;
    ...
}

@Entity
@IdClass(ItemId.class)
public class Item {
    @Id
    private Integer oid;
    @Id
    private Integer line;
    ...
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The following rules apply to composite primary keys:

- The primary key class must be public and must have a public no-arg constructor.
- The access type (field-based or property-based access) of a primary key class is determined by the access type of the entity for which it is the primary key, unless the primary key is an embedded ID and a different access type is specified.
- If property-based access is used, the properties of the primary key class must be public or protected.
- The primary key class must be serializable.
- The primary key class must define equals and hashCode methods. The semantics of value equality for these methods must be consistent with the database equality for the database types to which the key is mapped.
- A composite primary key must be either represented and mapped as an embeddable class or represented as an `id` class and mapped to multiple fields or properties of the entity class.
- If the composite primary key class is represented as an `id` class, the names of primary key fields or properties in the primary key class and those of the entity class to which the `id` class is mapped must correspond, and their types must be the same.

Example of an Embeddable composite key:

```
@Embeddable
public class ItemId {
    private Integer oid;
    private Integer line;
    public Integer getOid() {return oid;}
    public void setOid(Integer oid) {this.oid = oid;}
    public Integer getLine() {return line;}
    public void setLine(Integer line) {this.line = line;}
    public boolean equals(Object obj) {
        if (this == obj) {return true;}
        if (!(obj instanceof ItemId)) {return false;}
        ItemId item = (ItemId)obj;
        return oid.equals(item.oid) && line.equals(item.line);
    }
    public int hashCode() {
        final int prime = 31;
        int hash = 17;
        hash = hash*prime+this.oid.hashCode();
        hash = hash*prime+this.line.hashCode();
        return hash;
    }
}
```

Using Inheritance with Entities

Entities may use inheritance in different ways:

- Inherit behaviors and transient attributes
- Inherit behaviors and common state attributes using MappedSuperclass annotation
- Map both parent and child entities to database tables to represent different entity subtypes



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Using Unmapped Superclass

The purpose of an unmapped parent class is to contain behaviors and transient attributes.

- It is not itself mapped to any database table.
- It does not have to be abstract, but making it abstract would prevent its accidental instantiation.
- Nonentity classes cannot be passed as arguments to methods of the EntityManager or Query interfaces and cannot carry mapping information.

```
public abstract class TradeItem {  
    private float discount;  
    public void calculateDiscount() {...}  
}
```

```
@Entity  
public class Product extends TradeItem {  
    @Id  
    private Integer id;  
    private String name;  
    private float price;  
    private Date bestBefore;  
}
```

```
@Entity  
public class Service extends TradeItem {  
    @Id  
    private Integer id;  
    private String name;  
    private float price;  
    private String channel;  
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Using Mapped Superclass

The purpose of a mapped parent class is to contain behaviors and common attributes.

- It contains attributes that are mapped to columns of a table that corresponds to the inheriting entity subclass.
- Mapped superclasses cannot be passed as arguments to methods of the EntityManager or Query interfaces.
- A mapped superclass can be a source, but not the target of a persistent relationship.

```
@MappedSuperclass  
public class TradeItem {  
    @Id  
    private Integer id;  
    private String name;  
    private float price;  
    ...  
}  
  
@Entity  
public class Product extends TradeItem {  
    private Date bestBefore;  
    ...  
}  
  
@Entity  
public class Service extends TradeItem {  
    private String channel;  
    ...  
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Entity Inheritance Mapping Strategies

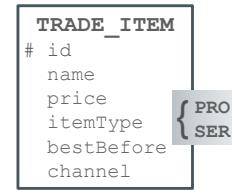
When mapping Entity hierarchy to database tables, you may use the following strategies:

- Single table: Default and most commonly used
- Joined: Less common, has performance issues
- Table per class: Extremely rare, has performance issues and may not be supported by JPA provider

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="itemType", length = 3,
                     discriminatorType=DiscriminatorType.STRING)
public class TradeItem {
    @Id
    private int id;
    private String name;
    private float price;
}

@Entity
@DiscriminatorValue(value = "PRO")
public class Product extends TradeItem {
    private Date bestBefore;
    ...
}
```

```
@Entity
@DiscriminatorValue(value = "SER")
public class Service extends TradeItem {
    private String channel;
    ...
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Inheritance may use different strategies described by the InheritanceType enum:

- **SINGLE_TABLE (default):** If you use this strategy, it is optional to annotate the root class with the `@Inheritance` annotation. With this strategy, all classes in the hierarchy are mapped to a single table in the database. This table has a discriminator column containing a value that identifies the subclass to which the instance represented by the row belongs.
- **JOINED:** The root of the class hierarchy is represented by a single table, and each subclass has a separate table that contains only those fields specific to that subclass. That is, the subclass table does not contain columns for inherited fields or properties. The subclass table also has a column or columns that represent its primary key, which is a foreign key to the primary key of the superclass table. This strategy provides good support for polymorphic relationships but requires one or more join operations to be performed when instantiating entity subclasses. This may result in poor performance for extensive class hierarchies. Similarly, queries that cover the entire class hierarchy require join operations between the subclass tables, resulting in decreased performance.
- Some Java Persistence API providers, including the default provider in GlassFish Server, require a discriminator column that corresponds to the root entity when using the joined subclass strategy. If you are not using automatic table creation in your application, make sure that the database table is set up correctly for the discriminator column defaults, or use the `@DiscriminatorColumn` annotation to match your database schema.

TABLE_PER_CLASS: Each concrete class is mapped to a separate table in the database. All fields or properties in the class, including inherited fields or properties, are mapped to columns in the class' table in the database. This strategy provides poor support for polymorphic relationships and usually requires either SQL UNION queries or separate SQL queries for each subclass for queries that cover the entire entity class hierarchy. Support for this strategy is optional and may not be supported by all Java Persistence API providers. The default Java Persistence API provider in GlassFish Server does not support this strategy.

By default, the discriminator column is a column named `DTYPE` with a string type.

The annotation `@DiscriminatorColumn` enables you to define the name of the column, its type, and any characteristics you want to assign to the type. For example, this discriminator column is identified as a string type with a maximum length of 20 characters.

Valid discriminator types include:

- `DiscriminatorType.STRING`
- `DiscriminatorType.INTEGER`
- `DiscriminatorType.CHAR`

The annotation `@DiscriminatorValue` determines the value stored in the discriminator column for the given entity type. If no `DiscriminatorValue` is defined for an entity in the hierarchy, the provider will determine the value of each entity type. If the discriminator type is `String`, the string name of the entity is used. If the discriminator type is `Integer`, either all of the entities need to be annotated with the `DiscriminatorValue` annotation, or none of them do.

Jairo Jose Silvera Sarmiento (jairo.silvera@gmail.com) has a
non-transferable license to use this Student Guide.

Batch and Concurrency APIs



ORACLE®



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Jairo Jose Silvera Sarmiento (jairo.silvera@gmail.com) has a
non-transferable license to use this Student Guide.

Objectives

This appendix covers the following topics:

- Concurrency in Java EE
 - Concurrency Utilities
 - Managed Executors
- Batch Applications for the Java Platform (JSR 352):
 - Describing batch jobs using JSL XML documents
 - Implementing batch jobs using JSR 352 API
 - Running and monitoring batch jobs



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



Concurrency

- Concurrency refers to the execution of two or more tasks at the same time.
- It enables you to take advantage of multicore processing and long wait times for external resources such as service calls or device access.
- Avoid creating programmatic threads or executors.
- Instead, know and use the EE concurrency mechanisms:
 - Message-Driven Beans
 - Do not require a reference to the actual EJB
 - Support fire and forget methods (no return value)
 - Covered in the lesson titled “Using Java Message Service”
 - Asynchronous EJBs
 - Require a reference to the actual EJB that can be injected by using CDI
 - Support return values in methods
 - Covered in the lesson titled “Implementing Business Logic by Using EJBs”
 - Concurrency utilities for Java EE
 - Provide maximum control over threads
 - Most native way of executing tasks
 - Covered in this Appendix



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

The Java Platform provides several ways to create concurrent execution:

- Creating thread classes and runnable interfaces
- Using the classes inside the `java.util.concurrent` package to create and manage executors and tasks
- Creating external processes that are executed separately from the JVM

Java is primarily concerned with the creation and management of threads. Threads are executed in the same process and can share information about the objects in the same JVM.

Using the Java Concurrency model allows you to execute multiple pieces of code at the same time to optimize performance.

Although threads optimize performance and provide better use of resources, there can be many situations where problems can occur, including deadlocks, thread starvation, concurrent access of resources, data inconsistency because of concurrent modification, and so on.

Java EE provides several mechanisms to run code asynchronously in a safe and managed way that allows the server to control the execution and if needed, stop it.

You should not create threads, executors, or processes because the server will not know about them and will not be able to stop them if needed.

If you use the EE concurrency mechanisms, the server is aware of the running tasks and if you need to stop the application, the server will be able to stop all running tasks and notify if such an event occurs.

Remember that Java EE applications run on the JVM of the Java EE Application Server. Therefore, your application should limit the use of resources that the Application Server may not know about. Examples of such resources include threads, file system accesses, sockets, swing windows, and so on.

In the following slides, you review the Java EE concurrency mechanisms.

Executor Service

- Inject an ExecutorService instance or get one by using JNDI.
 - ScheduledExecutorService: Execute task based on a schedule
 - ManagedExecutorService: Execute task once
 - ManagedScheduledExecutorService: Execute a task at some point in the future or schedule repeat execution of a task
- Execute or submit tasks:
 - Tasks can be Callable<V> or Runnable instances.
 - Tasks that are required to return a value must return a Future<V> instance

web.xml or ejb-jar.xml

```
<resource-env-ref>
  <description></description>
  <resource-env-ref-name>
    java:comp/env/concurrent/YourExecutor
  </resource-env-ref-name>
  <resource-env-ref-type>
    javax.enterprise.concurrent.ManagedExecutorService
  </resource-env-ref-type>
</resource-env-ref>
```



```
public class SomeClass {
  @Resource
  (name="java:comp/env/concurrent/YourExecutor")
  ManagedExecutorService mes;

  public void createTask() {
    Future<Integer> result = mes.submit(
      new Callable<Integer>() {
        public Integer call() { ... }
      }
    );
  }
}
```

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Managed Task Listener

ManagedTaskListener enables control and monitoring of a task.

```
public class MyTask implements Callable<Integer>, ManagedTaskListener {  
    public Integer call() throws Exception{...}  
    public void taskSubmitted(Future<?> f, ManagedExecutorService mes, Object t) {}  
    public void taskAborted(Future<?> f, ManagedExecutorService mes, Object t, Throwable e) {}  
    public void taskDone(Future<?> f, ManagedExecutorService mes, Object t, Throwable e) {}  
    public void taskStarting(Future<?> f, ManagedExecutorService mes, Object t) {}  
}  
  
public class SomeClass {  
    @Resource(name="java:comp/env/concurrent/YourExecutor")  
    ManagedExecutorService mes;  
    public void createTask(){  
        MyTask task = new MyTask();  
        Future<Integer> result = mes.submit(task);  
    }  
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

In the example in the slide, a class implements the Callable and ManagedTaskListener interfaces to add monitoring capabilities to a task.

You can use this event to start transactions, open resources, and better organize and manage your code.

A ManagedTaskListener will execute the events in the following manner:

- taskSubmitted: An event that is executed when the task is submitted to an executor
- taskStarting: An event that is executed when the task is about to run its call or run method
- taskAborted: An event that is executed if the task is cancelled or aborted due to the executor shutting down or the Future being cancelled
- taskDone: An event that is fired regardless of the outcome of the task and even if it could not even start. This is the last method executed in a task.

Batch Processing: Overview

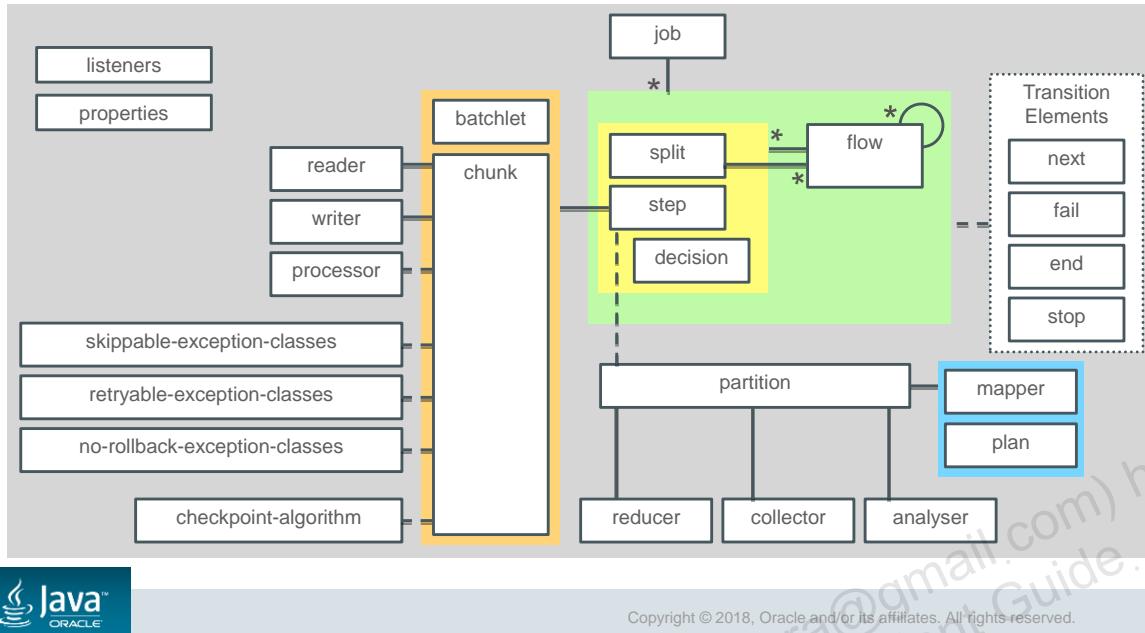
- Batch processing is the execution of a series of tasks.
 - Executes a series of tasks to process data
 - Is non-interactive. No user interaction is required.
 - Is usually long running and process intensive
 - Can be distributed to process data in many servers
- Batch applications are:
 - Suitable for noninteractive, bulk-oriented, and long-running tasks
- Examples include:
 - End-of-month bank statement generation
 - End-of-day jobs such as interest calculation
 - ETL (extract-transform-load) in a data warehouse
- Execution can be sequential, parallel, or decision-based.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

- The programming model for batch applications caters to batch processing concerns such as jobs, steps, repositories, reader processor writer patterns, chunks, checkpoints, parallel processing, flow, split, transactions, retries, sequencing, and partitioning.
- It standardizes batch processing for Java.
- The ability to execute batch jobs from Java EE is very important for many enterprise customers.
- It allows you to customize the handling of these jobs in terms of their individual steps. You can specify sequential or parallel execution as well as decisions that direct the execution path.
- The Batch API also provides for checkpointing and callback mechanisms.
- An individual batch job step may itself be a "chunk," whereby you can specify separately the handling of the input, processing, and output of the individual items that are part of the chunk.
- A step may be a "batchlet," which provides a more roll-your-own style of the step task's execution.

Job Specification Language (JSL) XML Structure



Job is a root element of the JSL XML document

```
<job id="{name}" restartable="{true|false}">
</job>
```

Inside a job element you include either one or more flow, or one or more split, or one or more step, or one or more decision element.

A flow element may contain either one or more flow, or one or more split, or one or more step, or one or more decision element.

A split should contain flow elements.

```
<flow id="{name}" next="{flow-id|step-id|split-id|decision-id}">
...
</flow>
```

```
<split id="{name}" next="{flow-id|step-id|split-id|decision-id}">
<flow> ... </flow>
...
</split>
```

```

<step id="{name}" start-limit="{integer}" allow-start-if-complete
      ="{true|false}" next="{flow-id|step-id|split-id|decision-id}">
  ...
<step>

  <decision id="{name}" ref="{ref-name}">
    ...
  </decision>

```

Transition element control execution order:

```

<next on="{exit status}" to="{step id|flow id|split id}" />
<fail on="{exit status}" exit-status="{exit status}" />
<end on="{exit status}" exit-status="{exit status}" />
<stop on="{exit status}" exit-status="{exit status}" restart="{step id | flow id | split id}" />

```

Inside the step you may place either a batchlet or a chunk element.

The batchlet element specifies a task-oriented batch step. It is specified as a child element of the step element. It is mutually exclusive with the chunk element. Steps of this type are useful for performing a variety of tasks that are not item-oriented, such as executing a command or doing file transfer.

```
<batchlet ref="{name}" />
```

The "chunk" element identifies a chunk type step. It is a child element of the step element. A chunk type step is periodically checkpointed by the batch runtime according to a configured checkpoint policy. Items processed between checkpoints are referred to as a "chunk". A single call is made to the ItemWriter per chunk. Each chunk is processed in a separate transaction. A chunk that is not complete is restartable from its last checkpoint. A chunk that is complete and belongs to a step configured with allow-start-if-complete=true runs from the beginning when restarted.

The "reader" element specifies the item reader for a chunk step. It is a child element of the "chunk" element. A chunk step must have one and only one item reader.

The "processor" element specifies the item processor for a chunk step. It is a child element of the "chunk" element. The processor element is optional on a chunk step. Only a single processor element may be specified.

The "writer" element specifies the item writer for a chunk step. It is a child element of the "chunk" element. A chunk type step must have one and only one item writer.

Checkpoint policy valid values are: "item" or "custom". The "item" policy means the chunk is checkpointed after a specified number of items are processed. The "custom" policy means the chunk is checkpointed according to a checkpoint algorithm implementation. Specifying "custom" requires that the checkpoint-algorithm element is also specified. It is an optional attribute. The default policy is "item".

By default, when any batch artefact that is part of a chunk type step throws an exception to the Batch Runtime, the job execution ends with a batch status of FAILED. The default behavior can be overridden for a reader, processor, or writer artefact by configuring exceptions to skip or to retry. The default behavior can be overridden for the entire step by configuring a transition element that matches the step's exit status.

```
<chunk checkpoint-policy="{item|custom}"
    item-count="{value}" time-limit="{value}"
    skip-limit="{value}" retry-limit="{value}">
    <reader ref="{name}" />
    <processor ref="{name}" />
    <writer ref="{name}" />
    <skippable-exception-classes>
        <include class="{class name}" />
        <exclude class="{class name}" />
    </skippable-exception-classes>
    <retryable-exception-classes>
        <include class="{class name}" />
        <exclude class="{class name}" />
    </retryable-exception-classes>
    <no-rollback-exception-classes>
        <include class="{class name}" />
        <exclude class="{class name}" />
    </no-rollback-exception-classes>
    <checkpoint-algorithm ref="{name}" />
</chunk>
```

A batch step may run as a partitioned step. A partitioned step runs as multiple instances of the same step definition across multiple threads, one partition per thread. The number of partitions and the number of threads is controlled through either a static specification in the Job XML or through a batch artefact called a partition mapper. Each partition needs the ability to receive unique parameters to instruct it which data on which to operate. Properties for each partition may be specified statically in the Job XML or through the optional partition mapper. Because each thread runs a separate copy of the step, chunking and checkpointing occur independently on each thread for chunk type steps.

There is an optional way to coordinate these separate units of work in a partition reducer so that backout is possible if one or more partitions experience failure. The PartitionReducer batch artefact provides a way to do that. A PartitionReducer provides programmatic control over logical unit of work demarcation that scopes all partitions of a partitioned step.

The partitions of a partitioned step may need to share results with a control point to decide the overall outcome of the step. The PartitionCollector and PartitionAnalyzer batch artefact pair provide for this need.

The "partition" element specifies that a step is a partitioned step. The partition element is a child element of the "step" element. It is an optional element.

A partition plan defines several configuration attributes that affect partitioned step execution. A partition plan specifies the number of partitions, the number of partitions to execute concurrently, and the properties for each partition. A partition plan may be defined in a Job XML declaratively or dynamically at runtime with a partition mapper.

The 'plan' element is a child element of the 'partition' element. The 'plan' element is mutually exclusive with partition mapper element.

The partition mapper provides a programmatic means for calculating the number of partitions and threads for a partitioned step. The partition mapper also specifies the properties for each partition. The mapper element specifies a reference to a PartitionMapper batch artefact. Note the mapper element is mutually exclusive with the plan element.

You can attach listeners and properties to almost all elements in this schema.

```
<listeners>
  <listener ref="{name}">
    ...
  </listeners>

  <properties>
    <property name="{property-name}" value="{name-value}" />
    ...
  </properties>
```

For more information on Job Specification Language refer to JSR-352 specification:
<https://jcp.org/aboutJava/communityprocess/final/jsr352/index.html>

Batch API Structure

- Batch job can be implemented as either:
 - Batchlet
 - or
 - Chunk
- Each chunk must have :
 - ItemReader
 - ItemWriter
- Each chunk may have:
 - ItemProcessor
- Other interfaces include:
 - CheckpointAlgorithm
 - Varios Listeners
 - Decider
- Implementation classes may inject JobContext object to access job properties.

```
public interface Batchlet {
    public String process(Object item) throws Exception;
    public void stop() throws Exception;
}
```

```
public interface ItemReader {
    public void open(Serializable checkpoint) throws Exception;
    public void close() throws Exception;
    public Object readItem() throws Exception;
    public Serializable checkpointInfo() throws Exception;
}
```

```
public interface ItemProcessor {
    public Object processItem(Object item) throws Exception;
}
```

```
public interface ItemWriter {
    public void open(Serializable checkpoint) throws Exception;
    public void close() throws Exception;
    public void writeItems(List items) throws Exception;
    public Serializable checkpointInfo() throws Exception;
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

CheckpointAlgorithm implements a custom checkpoint policy for a chunk step.

Batch API Listeners are:

- JobListener receives control before and after a job execution runs, and also if an exception is thrown during job processing.
- StepListener receives control before and after a step runs, and also if an exception is thrown during step processing.
- ChunkListener receives control at beginning and end of chunk processing and before and after a chunk checkpoint is taken.
- ItemReadListener receives control before and after an item is read by an item reader, and also if the reader throws an exception.
- ItemProcessListener receives control before and after an item is processed by an item processor, and also if the processor throws an exception.
- ItemWriteListener receives control before and after an item is written by an item writer, and also if the writer throws an exception.
- SkipListener receives control when a skippable exception is thrown from an item reader, processor, or writer.
- RetryProcessListener receives control when a retryable exception is thrown from an item reader, processor, or writer.

Batch jobs may be configured to run some of their steps in parallel. There are two supported parallelization models:

Partitioned

In the partitioned model, a step is configured to run as multiple instances across multiple threads. Each thread runs the same step or flow. This model is logically equivalent to launching multiple instances of the same step. It is intended that each partition processes a different range of the input items.

The partitioned model includes several optional batch artefacts to enable finer control over parallel processing:

- PartitionMapper provides a programmatic means for calculating the number of partitions and unique properties for each.
- PartitionReducer provides a unit of work demarcation around partition processing.
- PartitionCollector provides a means for merging interim results from individual partitions.
- PartitionAnalyzer provides a means to gather interim and final results from individual partitions for single point of control processing and decision making.

Concurrent

In the concurrent model, the flows defined by a split are configured to run concurrently on multiple threads, one flow per thread.

Decider may be used to determine batch exit status and sequencing between steps, splits, and flows in a Job XML. The decider returns a String value which becomes the exit status value on which the decision chooses the next transition. The Decider interface may be used to implement a Decider batch artefact.

Describe Job Using JSL XML Document

- Batch jobs are described using JSL XML documents stored in the `batch-jobs` directory under one of the following folders:
 - `META-INF` (in the ejb-jar deployment)
 - `WEB-INF/classes/META-INF` (in the war deployment)

ProductLoadJob.xml

```
<job id="ProductLoadJob"
  xmlns="http://xmlns.jcp.org/xml/ns/javaee version="1.0">
  <step id="ProcessProduct">
    <chunk item-count="10">
      <reader ref="ProductReader"/>
      <processor ref="ProductProcessor"/>
      <writer ref="ProductWriter"/>
    </chunk>
  </step>
</job>
```

- Batch jobs are implemented as Java classes that are implemented either as
 - or
 - Chunked (`Reader`,`Writer`,`Processor`)
 - Batchlet step types

```
@Named("ProductReader")
public class ProductReader
  implements ItemReader {
  ...
}

@Named("ProductProcessor")
public class ProductProcessor
  implements ItemProcessor {
  ...
}

@Named("ProductWriter")
public class ProductWriter
  implements ItemWriter {
  ...
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

SR 352 specifies a "Chunk Oriented" processing style as its primary pattern. Chunk-oriented processing refers to reading the data one item at a time, and creating "chunks" that will be written out, within a transaction boundary. One item is read in from an `ItemReader`, handed to an `ItemProcessor`, and aggregated. After the number of items read equals the commit interval, the entire chunk is written out via the `ItemWriter`, and then the transaction is committed.

The example in the slide above reads and processes products and aggregates them in groups of 10 to be passed to the writer.

Run Batch Job

- Batch jobs execution is triggered by the `JobOperator start` method.
- To start batch job, use its name as described by the JSL XML document.
- The properties object is used to substitute property values that could have been described in the JSL XML document.
- Job operator contains methods to control batch job, using operations such as:
 - `start`
 - `abandon`
 - `stop`
 - And so on
- The `JobExecution` object allows you to monitor a batch job, using operations such as:
 - `getBatchStatus`
 - `getCreateTime`
 - `getEndTime`
 - `getLastUpdatedTime`
 - `getStartTime`
 - And so on

```
JobOperator jo = BatchRuntime.getJobOperator();
Properties props = new Properties();
long jobId = jo.start("ProductLoadJob", props);

JobExecution je = jo.getJobExecution(jobId);
BatchStatus status = je.getBatchStatus();
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

A batch job must be initiated explicitly; for example, from a servlet or from an EJB timer or an EJB business method. A batch job is not automatically started when an application is deployed.

To run a job, you use the XML declaration for a job and the `JobOperator` class that is obtained from `BatchRuntime` to start the job and return the execution ID of the job. The execution ID can be used to monitor the execution of the job.

You can get basic information about the job, such as the start time, end time, and status of the batch process.

Jairo Jose Silvera Sarmiento (jairo.silvera@gmail.com) has a
non-transferable license to use this Student Guide.

JAXB API



ORACLE®



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Jairo Jose Silvera Sarmiento (jairo.silvera@gmail.com) has a
non-transferable license to use this Student Guide.

Objectives

This appendix covers the following topics:

- Creating Java to XML Schema mappings by using JAXB API
- Using JAXB API to marshal and unmarshal Java Objects



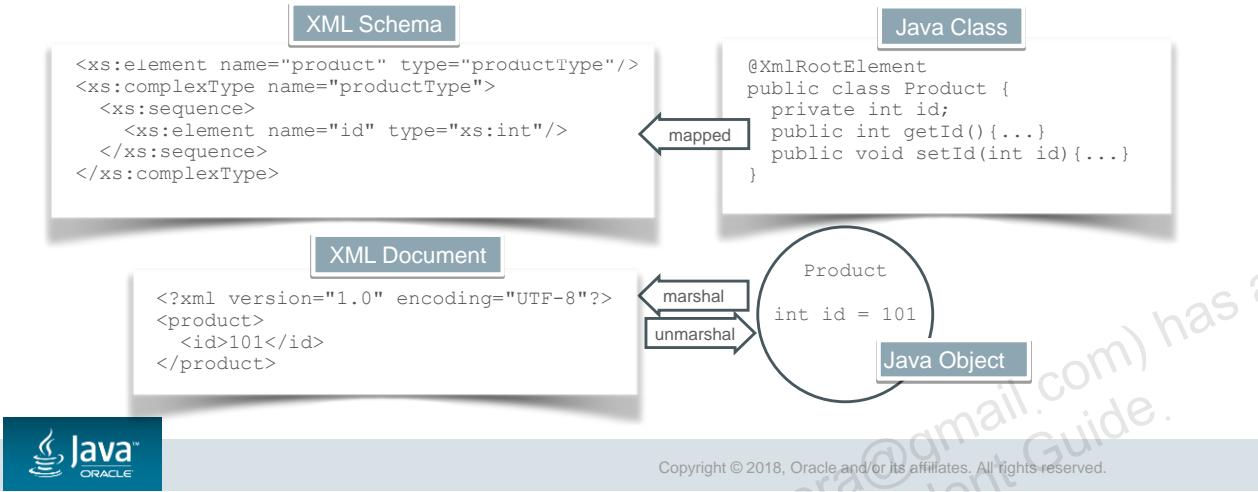
Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



JAXB API

Java Architecture for XML Binding (JAXB) API allows conversions between Java Objects and XML Documents.

- Used in various other APIs and Services
- Based on mappings between XML Schemas and Java Classes



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.



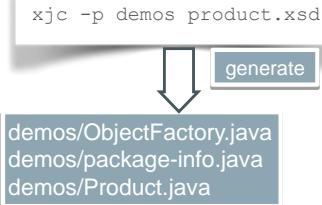
Bind Java Classes to XML Schema

Use the schemagen and xjc utilities to automate the binding of Java classes to XML mappings.

Generate XML schema based on Java class



Generate Java class based on XML schema



```

package demos;
@XmlRootElement
public class Product {
  private int id;
  public int getId(){...}
  public void setId(int id){...}
}
  
```

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Read and Write XML with JAXB

JAXB API provides classes to:

- Marshal (write) Java Object as XML Document:

```
try {
    Product p = new Product();
    p.setId(101);
    JAXBContext jc = JAXBContext.newInstance(Product.class);
    Marshaller m = jc.createMarshaller();
    OutputStream out = new FileOutputStream("product.xml");
    m.marshal(p, out);
} catch (JAXBException | IOException ex) {...}
```

- Unmarshal (read) XML Documents as Java Object:

```
try {
    JAXBContext jc = JAXBContext.newInstance(Product.class);
    Unmarshaller u = jc.createUnmarshaller();
    InputStream in = new FileInputStream("product.xml");
    Product p = (Product)u.unmarshal(in);
} catch (JAXBException | IOException ex) {...}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

JAXB Annotations Part I

JAXB Annotations control how exactly Java Object are represented as XML Structures.

- `XMLRootElement` maps Java class to XML Root Element.
- `XMLType` specifies a name for the XML Type and the order of elements.
- `XMLElement` controls binding of class members to XML.
- `XmlAttribute` maps a class member to an XML Attribute.
- `XMLTransient` indicates that the member should not be mapped to XML.

```
<xss:element name="product" type="productType"/>
<xss:complexType name="productType">
    <xss:sequence>
        <xss:element name="id" type="xs:int" minOccurs="1"/>
        <xss:element name="name" type="xs:string" minOccurs="0"/>
    </xss:sequence>
    <xss:attribute name="price" type="xs:float"/>
</xss:complexType>
```

```
<product price="1.99">
    <id>101</id>
    <name>Tea</name>
</product>
```

```
@XmlRootElement("product")
@XmlType(name="productType",
    propOrder={"id", "name"})

public class Product {
    @XmlElement(name="id", required=true)
    private int id;
    private String name;
    @XmlAttribute
    private float price;
    @XmlTransient
    private float discount;
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

`@XmlAccessorType` annotation on a class controls which members are bound to XML. Default value is `XmlAccessType.PUBLIC_MEMBER`

The possible values are:

`PUBLIC_MEMBER`: All public fields and public getter/setter method pairs are bound to XML elements.

`FIELD`: All fields, unless static or transient, are bound to XML elements.

`PROPERTY`: All getter/setter method pairs are bound to XML elements.

`NONE`: No members are bound to XML elements.

Using other annotations such as `@XmlValue`, `@XmlAttribute`, or `@XmlElement` instructs JAXB to map the member to XML even if the XML accessor type excludes the member from mapping.

JAXB Annotations Part II

- XMLValue maps a class member to simple content within a complex type or a simple type.
- XMLEnum enables Java enums to be mapped to XML enumerated values.

```

<xs:element name="product">
<xs:complexType>
  <xs:attribute name="price" type="xs:int"/>
  <xs:attribute name="state" type="productState"/>
</xs:complexType>
</xs:element>

<xs:simpleType name="productState">
  <xs:restriction base="xs:string">
    <xs:enumeration value="H"/>
    <xs:enumeration value="W"/>
    <xs:enumeration value="C"/>
  </xs:restriction>
</xs:simpleType>

<product id="101" state="H">Tea</product>

```

```

@XmlRootElement("product")
public class Product {
  @XmlAttribute
  private int id;
  @XmlAttribute
  ProductState state;
  @XmlValue
  private String name;
}

@XmlType
@XmlEnum
public enum ProductState {
  @XmlEnumValue("H")
  HOT,
  @XmlEnumValue("W")
  WARM,
  @XmlEnumValue("C")
  COLD;
}

```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Jairo Jose Silvera Sarmiento (jairo.silvera@gmail.com) has a
non-transferable license to use this Student Guide.

"Pre-CDI" Servlet Examples



ORACLE®



F

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Objectives

This appendix covers handling of "Old JavaEE style" pre-CDI examples of Request, Session, and Application scope.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Using Request Application Attributes Without CDI

You can store your objects in Request, Session, and Application scopes.

(Session scope is covered next.)

```
package demos;
public class SomeBean {
    public void doSomething() {
        //...
    }
}
```

Acquire object from the context.

Initialize object if necessary.

Use object.

Use set and get Attribute operations to store and retrieve objects:

```
protected void processRequest(HttpServletRequest request,
                               HttpServletResponse response)
                               throws ServletException, IOException {
    SomeBean sb1 = (SomeBean) request.getAttribute("SB");
    if (sb1 == null) {
        sb1 = new SomeBean();
        request.setAttribute("SB", sb1);
    }
    sb1.doSomething();
    ServletContext ctx = request.getServletContext();
    SomeBean sb2 = (SomeBean) ctx.getAttribute("SB");
    if (sb2 == null) {
        sb2 = new SomeBean();
        ctx.setAttribute("SB", sb2);
    }
    sb2.doSomething();
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Both Request and ServletContext object provide similar features on storing and retrieving attributes again corresponding scopes.

General use pattern is to try to get the relevant object out of corresponding context and check if it is has already been present in the corresponding context.

It is possible that object can be placed in to request session or application context by another web container component.

Using HttpSession Attributes Without CDI

```
package demos;
import java.io.Serializable;
public class SomeBean implements Serializable {
    public void doSomething() {
        //...
    }
}
```

Session-scoped objects should be serializable.

- Acquire HttpSession object.
- Initialize beans in a new session.
- Retrieve beans from an existing session.
- Use bean logic.

```
protected void processRequest(HttpServletRequest request,
                             HttpServletResponse response)
                             throws ServletException, IOException {
    SomeBean sb = null;
    HttpSession session = request.getSession();
    if (session.isNew()) {
        sb = new SomeBean();
        session.setAttribute("SB", sb);
    } else {
        sb = (SomeBean) session.getAttribute("SB");
    }
    sb.doSomething();
}
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

`HttpSession` provides a way to identify a user across more than one page request or visit to a website and to store information about that user.

The servlet container uses this interface to create a session between an HTTP client and an HTTP server. The session persists for a specified time period, across more than one connection or page request from the user. A session usually corresponds to one user, who may visit a site many times. The server can maintain a session in many ways such as using cookies or rewriting URLs (covered on the next page).

This interface allows servlets to:

- View and manipulate information about a session, such as the session identifier, creation time, and last accessed time
- Bind objects to sessions, allowing user information to persist across multiple user connections

A container can migrate sessions between VMs in a distributed container configuration.

A servlet should be able to handle cases in which the client does not choose to join a session, such as when cookies are intentionally turned off. Until the client joins the session, `isNew` returns true. If the client chooses not to join the session, `getSession` will return a different session on each request, and `isNew` will always return true.