



Red Hat Training and Certification

(ROLE)

EAP 7.0 AD183

Red Hat Application Development I: Programming in Java EE

Edition 3



Join a community dedicated to learning open source

The Red Hat® Learning Community is a collaborative platform for users to accelerate open source skill adoption while working with Red Hat products and experts.



Network with tens of thousands of community members



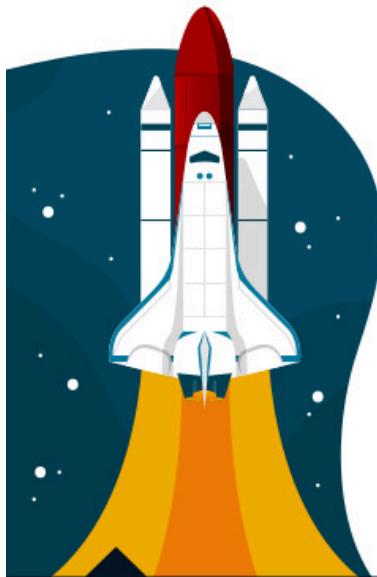
Engage in thousands of active conversations and posts



Join and interact with hundreds of certified training instructors



Unlock badges as you participate and accomplish new goals



This knowledge-sharing platform creates a space where learners can connect, ask questions, and collaborate with other open source practitioners.

Access free Red Hat training videos

Discover the latest Red Hat Training and Certification news

Connect with your instructor - and your classmates - before, after, and during your training course.

Join peers as you explore Red Hat products

Join the conversation learn.redhat.com



Copyright © 2020 Red Hat, Inc. Red Hat, Red Hat Enterprise Linux, the Red Hat logo, and Ansible are trademarks or registered trademarks of Red Hat, Inc. or its subsidiaries in the United States and other countries.

Red Hat Application Development I: Programming in Java EE



EAP 7.0 AD183

Red Hat Application Development I: Programming in Java EE

Edition 3 20220331

Publication date 20220331

Authors: Jim Rigsbee, Richard Allred, Zachary Guterman, Nancy K.A.N
Editor: Seth Kenlon

Copyright © 2018 Red Hat, Inc.

The contents of this course and all its modules and related materials, including handouts to audience members, are
Copyright © 2018 Red Hat, Inc.

No part of this publication may be stored in a retrieval system, transmitted or reproduced in any way, including, but not limited to, photocopy, photograph, magnetic, electronic or other record, without the prior written permission of Red Hat, Inc.

This instructional program, including all material provided herein, is supplied without any guarantees from Red Hat, Inc. Red Hat, Inc. assumes no liability for damages or legal action arising from the use or misuse of contents or details contained herein.

If you believe Red Hat training materials are being used, copied, or otherwise improperly distributed, please send email to training@redhat.com or phone toll-free (USA) +1 (866) 626-2994 or +1 (919) 754-3700.

Red Hat, Red Hat Enterprise Linux, the Red Hat logo, JBoss, OpenShift, Fedora, Hibernate, Ansible, CloudForms, RHCA, RHCE, RHCSA, Ceph, and Gluster are trademarks or registered trademarks of Red Hat, Inc. or its subsidiaries in the United States and other countries.

Linux® is the registered trademark of Linus Torvalds in the United States and other countries.

Java® is a registered trademark of Oracle American, Inc. and/or its affiliates.

XFS® is a registered trademark of Hewlett Packard Enterprise Development LP or its subsidiaries in the United States and/or other countries.

MySQL® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js® is a trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack word mark and the Square O Design, together or apart, are trademarks or registered trademarks of OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. Red Hat, Inc. is not affiliated with, endorsed by, or sponsored by the OpenStack Foundation or the OpenStack community.

All other trademarks are the property of their respective owners.

Contributors: Rob Locke, Bowe Strickland, Scott McBrien

Document Conventions	ix
Notes and Warnings	ix
Introduction	xi
Red Hat Application Development I: Programming in Java EE	xi
Orientation to the Classroom Environment	xii
Internationalization	xiv
1. Transitioning to Multi-tiered Applications	1
Describing Enterprise Applications	2
Quiz: Describing Enterprise Applications	4
Comparing Features of Java EE and Java SE	8
Quiz: Comparing Java EE and Java SE	12
Describing the Java Community Process	16
Quiz: Describing the Java Community Process (JCP)	19
Describing Multi-tiered Application Architecture	23
Quiz: Multi-tiered Application Architecture	28
Installing Java Development Tools	32
Workshop: Running the To Do List Application	37
Summary	44
2. Packaging and Deploying a Java EE Application	45
Describing an Application Server	46
Quiz: Describing an Application Server	50
Identifying JNDI Resources	54
Guided Exercise: Identifying JNDI Resources	57
Packaging and Deploying a Java EE Application	59
Workshop: Packaging and Deploying a Java EE Application	63
Lab: Packaging and Deploying Applications to an Application Server	72
Summary	79
3. Creating Enterprise Java Beans	81
Converting a POJO to an EJB	82
Guided Exercise: Creating a Stateless EJB	94
Accessing EJBs Locally and Remotely	101
Guided Exercise: Accessing an EJB Remotely	105
Describing the Life Cycle of an EJB	112
Quiz: The Lifecycle of an EJB	116
Demarcating Implicit and Explicit Transactions	120
Guided Exercise: Demarcating Transactions	126
Lab: Creating Enterprise Java Beans	131
Summary	137
4. Managing Persistence	139
Describing the Persistence API	140
Quiz: Describing the Persistence API	149
Persisting Data	151
Guided Exercise: Persisting Data	160
Annotating Classes to Validate Beans	168
Guided Exercise: Validating Data	174
Creating Queries	179
Guided Exercise: Creating Queries	186
Lab: Managing Persistence	191
Summary	199
5. Managing Entity Relationships	201
Configuring Entity Relationships	202
Guided Exercise: Configuring Entity Relationships	214

Describing Many-to-many Entity Relationships	222
Quiz: Describing Many-to-many Entity Relationships	226
Lab: Managing Entity Relationships	232
Summary	239
6. Creating REST Services	241
Describing Web Services Concepts	242
Quiz: Web Services	245
Creating REST Services with JAX-RS	247
Guided Exercise: Exposing a REST Service	255
Consuming a REST Service	262
Quiz: Consuming a REST Service	268
Lab: Creating REST Services	270
Summary	279
7. Implementing Contexts and Dependency Injection	281
Contrasting Dependency Injection and Resource Injection	282
Guided Exercise: Dependency Injection	286
Applying Contextual Scopes	292
Guided Exercise: Applying Scopes	295
Lab: Implementing Contexts and Dependency Injection	301
Summary	309
8. Creating Messaging Applications with JMS	311
Describing Messaging Concepts	312
Quiz: Describing Messaging Concepts	315
Describing JMS Architecture	317
Quiz: Describing JMS Architecture	323
Creating a JMS Client	325
Guided Exercise: Creating a JMS Client	330
Creating MDBs	335
Guided Exercise: Create a Message-driven Bean	338
Lab: Creating Messaging Applications with JMS	344
Summary	354
9. Securing Java EE Applications	355
Describing the JAAS Specification	356
Quiz: Describing the JAAS Specification	360
Configuring a Security Domain in JBoss EAP	362
Guided Exercise: Configuring a Security Domain in JBoss EAP	366
Securing a REST API	371
Guided Exercise: Securing a REST API	377
Lab: Securing Java EE Applications	384
Summary	398
10. Comprehensive Review: Red Hat Application Development I: Programming in Java EE	399
Comprehensive Review	400
Lab: Creating an API using JAX-RS	402
Lab: Persisting Data with JPA	420
Lab: Securing the REST API with JAAS	442

Document Conventions



Note

"Notes" are tips, shortcuts or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.



Important

"Important" boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring a box labeled "Important" will not cause data loss, but may cause irritation and frustration.



Warning

"Warnings" should not be ignored. Ignoring warnings will most likely cause data loss.



References

"References" describe where to find external documentation relevant to a subject.

Introduction

Red Hat Application Development I: Programming in Java EE

Red Hat Application Development I: Programming in Java EE (AD183) exposes experienced Java Standard Edition (Java SE) developers to the world of Java Enterprise Edition (Java EE). Students will learn about the various specifications that make up Java EE. Through hands-on labs, students will transform a simple Java SE command line application into a multi-tiered enterprise application using various Java EE specifications including Enterprise Java Beans, Java Persistence API, Java Messaging Service, JAX-RS for REST services, Contexts and Dependency Injection, and security.

Objectives

- Describe the differences between Java SE and Java EE application architectures.
- Create application components using the EJB, JPA, JAX-RS, CDI, JMS, and JAAS specifications.
- Develop back-end components necessary to support a three-tiered web application using JBoss and Maven tooling.
- Deploy applications to Red Hat JBoss Enterprise Application Platform.

Audience

- Developers with Java SE experience.

Prerequisites

- Proficient in developing Java SE applications.
- Proficient in using an IDE such as Red Hat Developer Studio or Eclipse.
- Experience with Maven is recommended.

Orientation to the Classroom Environment

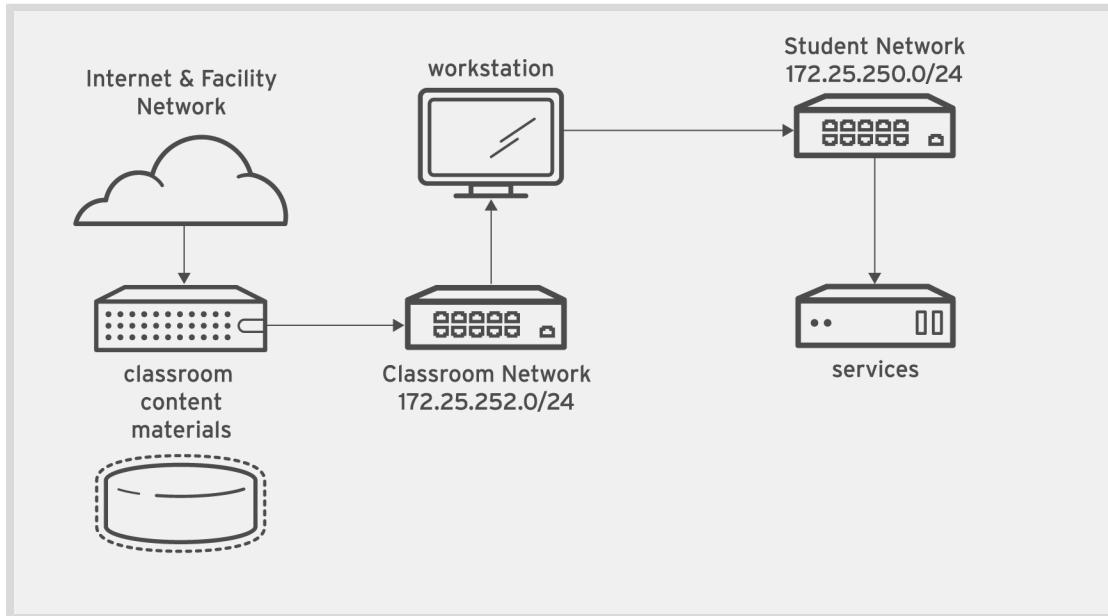


Figure 0.1: Classroom Environment

In this course, the main computer system used for hands-on learning activities is **workstation**. One other machine, **services**, will also be used by students for these activities. Both systems are in the `lab.example.com` DNS domain.

All student computer systems have a standard user account, **student**, with the password **student**. The root password on all student systems is **redhat**.

Classroom Machines

Machine name	IP addresses	Role
<code>workstation.lab.example.com</code>	172.25.250.254	Graphical workstation used for system administration
<code>services.lab.example.com</code>	172.25.250.10	Hosts the Nexus repository

One additional function of **workstation** is that it acts as a router between the network that connects the student machines and the classroom network. If **workstation** is down, other student machines are only able to access systems on the student network.

There are several systems in the classroom that provide supporting services. Two servers, `content.example.com` and `materials.example.com`, are sources for software and lab materials used in hands-on activities. Information on how to use these servers will be provided in the instructions for those activities.

Controlling Your Station

The top of the console describes the state of your machine.

Machine States

State	Description
none	Your machine has not yet been started. When started, your machine will boot into a newly initialized state (the disk will have been reset).
starting	Your machine is in the process of booting.
running	Your machine is running and available (or, when booting, soon will be.)
stopping	Your machine is in the process of shutting down.
stopped	Your machine is completely shut down. Upon starting, your machine boots into the same state as when it was shut down (the disk has been preserved).
impaired	A network connection to your machine cannot be made. Typically this state is reached when a student has corrupted networking or firewall rules. If the condition persists after a machine reset, or is intermittent, please open a support case.

Depending on the state of your machine, a selection of the following actions is available to you.

Machine Actions

Action	Description
Start Station	Start ("power on") the machine.
Stop Station	Stop ("power off") the machine, preserving the contents of its disk.
Reset Station	Stop ("power off") the machine, resetting the disk to its initial state. Caution: Any work generated on the disk will be lost.
Refresh	Refreshing the page probes the machine state.
Increase Timer	Adds 15 minutes to the timer for each click.

The Station Timer

Your Red Hat Online Learning enrollment entitles you to a certain amount of computer time. In order to help you conserve your time, the machines have an associated timer, which is initialized to 60 minutes when your machine is started.

The timer operates as a "dead man's switch," which decrements as your machine is running. If the timer is winding down to 0, you may choose to increase the timer.

Internationalization

Language Support

Red Hat Enterprise Linux 7 officially supports 22 languages: English, Assamese, Bengali, Chinese (Simplified), Chinese (Traditional), French, German, Gujarati, Hindi, Italian, Japanese, Kannada, Korean, Malayalam, Marathi, Odia, Portuguese (Brazilian), Punjabi, Russian, Spanish, Tamil, and Telugu.

Per-user Language Selection

Users may prefer to use a different language for their desktop environment than the system-wide default. They may also want to set their account to use a different keyboard layout or input method.

Language Settings

In the GNOME desktop environment, the user may be prompted to set their preferred language and input method on first login. If not, then the easiest way for an individual user to adjust their preferred language and input method settings is to use the Region & Language application. Run the command `gnome-control-center region`, or from the top bar, select (User) > Settings. In the window that opens, select Region & Language. The user can click the Language box and select their preferred language from the list that appears. This will also update the Formats setting to the default for that language. The next time the user logs in, these changes will take full effect.

These settings affect the GNOME desktop environment and any applications, including `gnome-terminal`, started inside it. However, they do not apply to that account if accessed through an ssh login from a remote system or a local text console (such as `tty2`).



Note

A user can make their shell environment use the same LANG setting as their graphical environment, even when they log in through a text console or over ssh. One way to do this is to place code similar to the following in the user's `~/.bashrc` file. This example code will set the language used on a text login to match the one currently set for the user's GNOME desktop environment:

```
i=$(grep 'Language=' /var/lib/AccountService/users/${USER} \
    | sed 's/Language=//')
if [ "$i" != "" ]; then
    export LANG=$i
fi
```

Japanese, Korean, Chinese, or other languages with a non-Latin character set may not display properly on local text consoles.

Individual commands can be made to use another language by setting the LANG variable on the command line:

```
[user@host ~]$ LANG=fr_FR.utf8 date  
jeu. avril 24 17:55:01 CDT 2014
```

Subsequent commands will revert to using the system's default language for output. The `Locale` command can be used to check the current value of `LANG` and other related environment variables.

Input Method Settings

GNOME 3 in Red Hat Enterprise Linux 7 automatically uses the IBus input method selection system, which makes it easy to change keyboard layouts and input methods quickly.

The Region & Language application can also be used to enable alternative input methods. In the Region & Language application's window, the **Input Sources** box shows what input methods are currently available. By default, **English (US)** may be the only available method. Highlight **English (US)** and click the **Keyboard** icon to see the current keyboard layout.

To add another input method, click the **+** button at the bottom left of the **Input Sources** window. An **Add an Input Source** window will open. Select your language, and then your preferred input method or keyboard layout.

Once more than one input method is configured, the user can switch between them quickly by typing **Super+Space** (sometimes called **Windows+Space**). A *status indicator* will also appear in the GNOME top bar, which has two functions: It indicates which input method is active, and acts as a menu that can be used to switch between input methods or select advanced features of more complex input methods.

Some of the methods are marked with gears, which indicate that those methods have advanced configuration options and capabilities. For example, the Japanese **Japanese (Kana Kanji)** input method allows the user to pre-edit text in Latin and use **Down Arrow** and **Up Arrow** keys to select the correct characters to use.

US English speakers may find also this useful. For example, under **English (United States)** is the keyboard layout **English (international AltGr dead keys)**, which treats **AltGr** (or the right **Alt**) on a PC 104/105-key keyboard as a "secondary-shift" modifier key and dead key activation key for typing additional characters. There are also Dvorak and other alternative layouts available.



Note

Any Unicode character can be entered in the GNOME desktop environment if the user knows the character's Unicode code point, by typing **Ctrl+Shift+U**, followed by the code point. After **Ctrl+Shift+U** has been typed, an underlined **u** will be displayed to indicate that the system is waiting for Unicode code point entry.

For example, the lowercase Greek letter lambda has the code point **U+03BB**, and can be entered by typing **Ctrl+Shift+U**, then **03bb**, then **Enter**.

System-wide Default Language Settings

The system's default language is set to US English, using the UTF-8 encoding of Unicode as its character set (`en_US.utf8`), but this can be changed during or after installation.

From the command line, `root` can change the system-wide locale settings with the `localectl` command. If `localectl` is run with no arguments, it will display the current system-wide locale settings.

Introduction

To set the system-wide language, run the command `localectl set-locale LANG=locale`, where `locale` is the appropriate `$LANG` from the "Language Codes Reference" table in this chapter. The change will take effect for users on their next login, and is stored in `/etc/locale.conf`.

```
[root@host ~]# localectl set-locale LANG=fr_FR.utf8
```

In GNOME, an administrative user can change this setting from Region & Language and clicking the **Login Screen** button at the upper-right corner of the window. Changing the **Language** of the login screen will also adjust the system-wide default language setting stored in the `/etc/locale.conf` configuration file.

**Important**

Local text consoles such as `tty2` are more limited in the fonts that they can display than `gnome-terminal` and `ssh` sessions. For example, Japanese, Korean, and Chinese characters may not display as expected on a local text console. For this reason, it may make sense to use English or another language with a Latin character set for the system's text console.

Likewise, local text consoles are more limited in the input methods they support, and this is managed separately from the graphical desktop environment. The available global input settings can be configured through `localectl` for both local text virtual consoles and the X11 graphical environment. See the `localectl(1)`, `kbd(4)`, and `vconsole.conf(5)` man pages for more information.

Language Packs

When using non-English languages, you may want to install additional "language packs" to provide additional translations, dictionaries, and so forth. To view the list of available langpacks, run `yum langavailable`. To view the list of langpacks currently installed on the system, run `yum langlist`. To add an additional langpack to the system, run `yum langinstall code`, where `code` is the code in square brackets after the language name in the output of `yum langavailable`.

**References**

`locale(7)`, `localectl(1)`, `kbd(4)`, `locale.conf(5)`, `vconsole.conf(5)`, `unicode(7)`, `utf-8(7)`, and `yum-langpacks(8)` man pages

Conversions between the names of the graphical desktop environment's X11 layouts and their names in `localectl` can be found in the file `/usr/share/X11/xkb/rules/base.lst`.

Language Codes Reference

Language Codes

Language	\$LANG value
English (US)	en_US.utf8
Assamese	as_IN.utf8

Language	\$LANG value
Bengali	bn_IN.utf8
Chinese (Simplified)	zh_CN.utf8
Chinese (Traditional)	zh_TW.utf8
French	fr_FR.utf8
German	de_DE.utf8
Gujarati	gu_IN.utf8
Hindi	hi_IN.utf8
Italian	it_IT.utf8
Japanese	ja_JP.utf8
Kannada	kn_IN.utf8
Korean	ko_KR.utf8
Malayalam	ml_IN.utf8
Marathi	mr_IN.utf8
Odia	or_IN.utf8
Portuguese (Brazilian)	pt_BR.utf8
Punjabi	pa_IN.utf8
Russian	ru_RU.utf8
Spanish	es_ES.utf8
Tamil	ta_IN.utf8
Telugu	te_IN.utf8

Chapter 1

Transitioning to Multi-tiered Applications

Overview

Goal

Describe Java EE features and distinguish between Java EE and Java SE applications.

Objectives

- Describe "enterprise application" and name some of the benefits of Java EE applications.
- Compare the features of Java EE to Java SE.
- Describe the specifications and version numbers for Java EE 7 and the process used to introduce new and updated APIs into Java EE.
- Describe various multi-tiered architectures.
- Install JBoss Developer Studio, Maven, and JBoss Enterprise Application Platform.

Sections

- Describing Enterprise Applications (and Quiz)
- Comparing Features of Java EE and Java SE (and Quiz)
- Describing the Java Community Process (and Quiz)
- Describing Multi-tiered Application Architecture (and Quiz)
- Installing Java Development Tools (and Guided Exercise)

Describing Enterprise Applications

Objectives

After completing this section, students should be able to describe the basic concepts and benefits of enterprise applications.

Enterprise Applications

An *enterprise application* is a software application typically used in large business organizations. Enterprise applications often provide the following features:

- Support for concurrent users and external systems.
- Support for synchronous and asynchronous communication using different protocols.
- Ability to handle transactional workloads that coordinate between data repositories such as queues and databases.
- Support for scalability to handle future growth.
- A resilient and distributed platform to ensure high availability.
- Support for highly secure access control for different types of users.
- Ability to integrate with back-end systems and web services.

Typical examples of enterprise applications include Enterprise Resource Planning (ERP), Customer Relationship Management (CRM), Content Management Systems (CMS), e-commerce systems, internet and intranet portals.

Benefits of Java EE Enterprise applications

Java Enterprise Edition (Java EE) is a specification for developing enterprise applications using Java. It is a platform-independent standard that is developed under the guidance of the Java Community Process (JCP).

The Java EE 7 specification consists of a number of component application programming interfaces (API) that are implemented by an *application server*. The Red Hat JBoss Enterprise Application Platform (EAP), which you will use in this course, implements the Java EE standard.

The benefits of developing Java EE based enterprise applications are:

- Platform-independent applications can be developed and will run on many different types of operating systems (on small PCs as well as large mainframes).
- Applications are portable across Java EE compliant application servers due to the Java EE standard.
- The Java EE specification provides a large number of APIs typically used by enterprise applications such as web services, asynchronous messaging, transactions, database connectivity, thread pools, batching utilities, and security. There is no need to develop these components manually, thereby reducing development time.

- A large number of third-party, ready-to-use applications and components that target specific domains such as finance, insurance, telecom, and other industries are certified to run and integrate with Java EE application servers.
- A large number of sophisticated tools such as IDEs, monitoring systems, enterprise application integration (EAI) frameworks, and performance measurement tools are available for Java EE applications from third-party vendors.



References

JCP

<https://www.jcp.org/en/home/index>

► Quiz

Describing Enterprise Applications

Choose the correct answers to the following questions:

- ▶ 1. **Which of the following two statements can be considered an enterprise application? (Choose two.)**
 - a. An online banking system for a bank with millions of customers.
 - b. A program that calculates the factorial of numbers between 1 and 100,000.
 - c. A real-time embedded system that controls a remote satellite.
 - d. An online payment gateway for a credit card company that processes millions of transactions per day.
 - e. A program that simulates a 3D representation of an aircraft to study the impact of turbulence on aircraft of different shapes and sizes.

- ▶ 2. **Which of the following two statements about the Java EE specification and application servers are correct? (Choose two.)**
 - a. There are different versions of the specification for different operating systems.
Applications should be implemented differently on each operating system leveraging the operating system specific features.
 - b. Although an application is implemented to be fully Java EE compliant, you need to re-implement certain features and recompile the application when deploying it on different Java EE compliant servers.
 - c. A fully Java EE compliant application can be deployed on different Java EE compliant servers without recompiling and re-implementing features.
 - d. A Java EE compliant application server provides facilities for asynchronous messaging.
 - e. A Java EE compliant application server does not provide thread pooling features by default. This feature has to be manually implemented by the developer.

- ▶ 3. **Which statement describing a Java EE compliant application server is correct?**
 - a. The ability to create web services (SOAP, REST) are not provided by default.
 - b. No transaction management facilities are available. All transactions have to be manually managed by the developer.
 - c. The application server provides automatic transaction management. If required, the developer can write code to also manage transactions manually.
 - d. No database connectivity API (JPA) is provided by default. Third-party external libraries have to be used for connecting to databases.

- 4. A company named **ABC Inc** is migrating a large and complex legacy mainframe application written in COBOL™ to the Java EE platform. Which of the following three features can be leveraged from a Java EE compliant application server without manually implementing them? (Choose three.)
- a. Database connectivity to an RDBMS (which is JDBC compliant).
 - b. A utility that reads and writes EBCDIC encoded files to and from a legacy mainframe system with a proprietary file system.
 - c. Role-based security.
 - d. Batch operations for scheduled execution of a reporting application, that generates reports from an RDBMS on a daily, monthly and quarterly basis.
 - e. A custom adapter to communicate with a remote legacy hierarchical database management system that is not JDBC compliant.

► Solution

Describing Enterprise Applications

Choose the correct answers to the following questions:

- ▶ 1. **Which of the following two statements can be considered an enterprise application? (Choose two.)**
 - a. An online banking system for a bank with millions of customers.
 - b. A program that calculates the factorial of numbers between 1 and 100,000.
 - c. A real-time embedded system that controls a remote satellite.
 - d. An online payment gateway for a credit card company that processes millions of transactions per day.
 - e. A program that simulates a 3D representation of an aircraft to study the impact of turbulence on aircraft of different shapes and sizes.

- ▶ 2. **Which of the following two statements about the Java EE specification and application servers are correct? (Choose two.)**
 - a. There are different versions of the specification for different operating systems.
Applications should be implemented differently on each operating system leveraging the operating system specific features.
 - b. Although an application is implemented to be fully Java EE compliant, you need to re-implement certain features and recompile the application when deploying it on different Java EE compliant servers.
 - c. A fully Java EE compliant application can be deployed on different Java EE compliant servers without recompiling and re-implementing features.
 - d. A Java EE compliant application server provides facilities for asynchronous messaging.
 - e. A Java EE compliant application server does not provide thread pooling features by default. This feature has to be manually implemented by the developer.

- ▶ 3. **Which statement describing a Java EE compliant application server is correct?**
 - a. The ability to create web services (SOAP, REST) are not provided by default.
 - b. No transaction management facilities are available. All transactions have to be manually managed by the developer.
 - c. The application server provides automatic transaction management. If required, the developer can write code to also manage transactions manually.
 - d. No database connectivity API (JPA) is provided by default. Third-party external libraries have to be used for connecting to databases.

- 4. A company named **ABC Inc** is migrating a large and complex legacy mainframe application written in COBOL™ to the Java EE platform. Which of the following three features can be leveraged from a Java EE compliant application server without manually implementing them? (Choose three.)
- a. Database connectivity to an RDBMS (which is JDBC compliant).
 - b. A utility that reads and writes EBCDIC encoded files to and from a legacy mainframe system with a proprietary file system.
 - c. Role-based security.
 - d. Batch operations for scheduled execution of a reporting application, that generates reports from an RDBMS on a daily, monthly and quarterly basis.
 - e. A custom adapter to communicate with a remote legacy hierarchical database management system that is not JDBC compliant.

Comparing Features of Java EE and Java SE

Objectives

After completing this section, students should be able to:

- Describe the differences between Java EE and Java SE.
- Run the application from the command line.
- Preview the web application.

Comparing Java Enterprise Edition (Java EE) and Java SE

When you install the Java Development Kit (JDK) for your operating system, it provides the compiler, debugger, tools, and runtime environment for hosting your Java applications, the Java Virtual Machine (JVM), and a large set of reusable component classes that are commonly used by applications. This application programming interface (API) provides packages and classes for networking, I/O, XML parsing, database connectivity, developing graphical user interfaces (GUI), and many more. This API is commonly known as the *Java Standard Edition (Java SE)*.

Java SE is generally used to develop stand-alone programs, tools, and utilities that are mainly run from the command line, GUI programs, and server processes that need to run as daemons (that is, programs that run continuously in the background until they are stopped).

The Java EE specification is a set of APIs built on top of Java SE. It provides a runtime environment for running multi-threaded, transactional, secure and scalable enterprise applications. It is important to note that unlike Java SE, Java EE is mainly a set of standard specifications for an API, and runtime environments that implement these APIs are generally called as *application servers*.

An application server that passes a test suite called the *Technology Compatibility Kit (TCK)* for Java EE is known as a *Java EE compliant* application server. There are different versions of Java EE. While new APIs and features are incrementally added in each new version, compatibility with earlier versions is strictly maintained.

Java EE includes support for multiple *profiles*, or subsets of APIs. For example, the Java EE 7 specification defines two profiles: the *full profile* and the *web profile*.

The Java EE 7 **web profile** is designed for web application development and supports a subset of the APIs defined by Java EE 7 related web-based technologies.

The Java EE 7 **full profile** contains all APIs defined by Java EE 7 (including all the items in the web profile). When developing EJBs, messaging applications, and web services (in contrast to web applications), you should use the full profile.

A Java EE 7 compliant application server, such as **Red Hat JBoss Enterprise Application Platform (EAP)**, implements both profiles and provides a number of APIs that are commonly used in enterprise applications, including:

- Batch API

- Java API for JSON Processing (JSON-P)
- Concurrency utilities
- WebSocket API
- Java Messaging Service (JMS)
- Java Persistence API (JPA)
- Java Connector Architecture (JCA)
- Java API for RESTful web services (JAX-RS)
- Java API for XML web services (JAX-WS)
- Servlet API
- Java Server Faces (JSF)
- Java Server Pages (JSP)
- Contexts and Dependency Injection (CDI)
- Java Transaction API (JTA)
- Enterprise Java Beans (EJB)
- Bean Validation API

Building, Packaging and Deploying Java SE and Java EE Applications

For relatively simple standalone Java SE applications, the code can be built, packaged, and run on the command line by using the compiler and runtime tools (`java`, `javac`, `jar`, `jdb`, and so on) that are part of the JDK. Several mature Integrated Development Environments (IDEs), such as *Red Hat JBoss Developer Studio (JBDS)* or *Eclipse*, are used to simplify the building and packaging process.

The preferred way to ship standalone Java applications in a platform-neutral way is to package the application as a *Java Archive (JAR)* file. JAR files can optionally be made executable by adding *manifest* entries (a plain text file packaged alongside the Java classes inside the JAR file) to the JAR file to indicate the main runnable class.

Java EE applications consist of multiple components that depend on a large number of JAR files that are required at runtime. The deployment process for Java EE applications is different. Java EE applications are deployed on a Java EE compatible application server and these deployments can be of different types:

- **JAR files:** Individual modules of an application and Enterprise Java Beans (EJBs) can be deployed as separate JAR files. Third-party libraries and frameworks are also packaged as JAR files. If your application depends on these libraries, the library JAR files should be deployed on the application server. JAR files have a `.jar` extension.
- **Web Archive (WAR) files:** If your Java EE application has a web-based front end or is providing RESTful service endpoints, then code and assets related to the web front end and the services can be packaged as a *WAR* file. A WAR file has a `.war` extension and is essentially

a compressed file containing code, static HTML, images, CSS, and JS assets, as well as XML deployment descriptor files along with dependent JAR files packaged inside it.

- **Enterprise Archive (EAR) files:** An EAR file has an extension of .ear and is essentially a compressed file with one or more WAR or JAR files and some XML deployment descriptors inside it. It is useful in scenarios where the application contains multiple WAR files or reuses some common JAR files across modules. In such cases, it is easier to deploy and manage the application as a single deployable unit.

It is also a best practice to use a build tool such as *Apache Maven* to simplify building, packaging, testing, executing, and deploying Java SE and Java EE applications. Maven has a plug-in architecture to extend its core functionality.

There are Maven plug-ins for building, packaging, and deploying Java EE applications. All deployment types are supported. Maven can also deploy and undeploy applications to and from JBoss EAP without restarting the application server. Integrated Development Environments (IDEs) such as JBoss Developer Studio (JBDS) also have native support for Maven built-in by default. All Maven tasks can be run from within JBDS itself without using the command line.

To run a stand-alone application that uses only the Java SE API such as, for example, the command-line based To Do List Application that is packaged as a JAR file, you can use the `java -jar` command:

```
[student@workstation todojse]$ java -jar todojse-1.0.jar
```

```
[N]ew | [C]omplete | [R]ead | [D]elete | [L]ist | [Q]uit:  
N  
Enter item description:  
task 1  
Is todo item completed? [Y/N]:  
N  
Do you want to add more items[Y/N]  
N  
Successfully added new todo item:  
  
[N]ew | [C]omplete | [R]ead | [D]elete | [L]ist | [Q]uit:  
L  
---- There are 1 items in the list ----  
1 - task 1 - PENDING
```

Figure 1.1: The Stand-alone Java SE-based To Do List Application

In contrast, the web-based version is deployed to the Java EE compliant *application server*. The example To Do List Application is packaged as a WAR file that is deployed to an application server like EAP.

If an older version of the WAR file is already deployed, the old version is undeployed and the new version is deployed without restarting the application server. Such a process is called *hot deployment* and is used extensively during development, testing, as well as in production rollouts.

Task List

ID	Description	Done	
1	Pick up newspaper	<input type="checkbox"/>	
2	Buy groceries	<input checked="" type="checkbox"/>	
3	Pay telephone bill	<input type="checkbox"/>	
4	Buy milk	<input type="checkbox"/>	
5	Buy butter	<input checked="" type="checkbox"/>	



Figure 1.2: Java EE 7 web-based To Do List Application task list

Add Task

Description:

Read a Book

Completed:



Save

Clear

Figure 1.3: Add a task in the To Do List Application



References

Java EE 7 Specification JSR

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

Red Hat JBoss EAP 7 Release Notes

<https://access.redhat.com/documentation/en/red-hat-jboss-enterprise-application-platform/version-7.0>

► Quiz

Comparing Java EE and Java SE

Choose the correct answers to the following questions:

► 1. **Which of the following APIs is part of the Java EE 7 specification?**

- a. Java Database Connectivity (JDBC)
- b. Java Server Faces (JSF)
- c. Java Cryptography Extensions (JCE)
- d. Java Swing
- e. Java Security

► 2. **Which of the following two statements about the Java EE 7 specification are correct? (Choose two.)**

- a. Java EE 7 is a specification and not an implementation that can be used directly.
- b. There must be separate Java EE 7 implementations for different operating systems.
- c. The Java EE 7 standard defines two profiles: *web* and *full*.
- d. The Java EE 7 standard defines four profiles: *small*, *medium*, *web*, and *full*.

► 3. **Which of the following two statements about the Java SE are correct? (Choose two.)**

- a. Java SE is used to develop stand-alone applications, like tools, utilities, and GUI applications that can run from the command line.
- b. There must be separate Java SE implementations for different operating systems.
- c. The Java SE can be used to write an application to store data in the database.
- d. The Java SE cannot be used to write multi-threaded applications.

► 4. **Which of the following two statements about the deployment types in Java EE 7 are correct? (Choose two.)**

- a. Web applications are typically packaged as JAR files for deployment to an application server.
- b. Web applications are typically packaged as WAR files for deployment to an application server.
- c. A WAR file can contain EAR and JAR files as well as deployment descriptors.
- d. An EAR file can contain WAR files, JAR files, and deployment descriptors.
- e. An EAR file cannot be directly deployed inside an application server. You have to deploy the WAR and JAR files inside it separately.

► 5. Which of the following two statements about Apache Maven are correct? (Choose two.)

- a. Maven can be used to build, package, and test both Java EE and Java SE applications.
- b. Maven can only be used to build, package, and test Java EE applications. Maven cannot be used for building, packaging, and testing Java SE applications.
- c. Maven cannot deploy and undeploy applications to and from JBoss EAP. You have to manually restart the application server after every deployment and undeployment.
- d. Maven can automatically deploy and undeploy applications from JBoss EAP. There is no need to restart the application server after every deployment and undeployment.
- e. There is no IDE support for Maven tasks. All Maven tasks have to be invoked from the command line.

► Solution

Comparing Java EE and Java SE

Choose the correct answers to the following questions:

► 1. **Which of the following APIs is part of the Java EE 7 specification?**

- a. Java Database Connectivity (JDBC)
- b. Java Server Faces (JSF)
- c. Java Cryptography Extensions (JCE)
- d. Java Swing
- e. Java Security

► 2. **Which of the following two statements about the Java EE 7 specification are correct? (Choose two.)**

- a. Java EE 7 is a specification and not an implementation that can be used directly.
- b. There must be separate Java EE 7 implementations for different operating systems.
- c. The Java EE 7 standard defines two profiles: *web* and *full*.
- d. The Java EE 7 standard defines four profiles: *small*, *medium*, *web*, and *full*.

► 3. **Which of the following two statements about the Java SE are correct? (Choose two.)**

- a. Java SE is used to develop stand-alone applications, like tools, utilities, and GUI applications that can run from the command line.
- b. There must be separate Java SE implementations for different operating systems.
- c. The Java SE can be used to write an application to store data in the database.
- d. The Java SE cannot be used to write multi-threaded applications.

► 4. **Which of the following two statements about the deployment types in Java EE 7 are correct? (Choose two.)**

- a. Web applications are typically packaged as JAR files for deployment to an application server.
- b. Web applications are typically packaged as WAR files for deployment to an application server.
- c. A WAR file can contain EAR and JAR files as well as deployment descriptors.
- d. An EAR file can contain WAR files, JAR files, and deployment descriptors.
- e. An EAR file cannot be directly deployed inside an application server. You have to deploy the WAR and JAR files inside it separately.

► **5. Which of the following two statements about Apache Maven are correct? (Choose two.)**

- a. Maven can be used to build, package, and test both Java EE and Java SE applications.
- b. Maven can only be used to build, package, and test Java EE applications. Maven cannot be used for building, packaging, and testing Java SE applications.
- c. Maven cannot deploy and undeploy applications to and from JBoss EAP. You have to manually restart the application server after every deployment and undeployment.
- d. Maven can automatically deploy and undeploy applications from JBoss EAP. There is no need to restart the application server after every deployment and undeployment.
- e. There is no IDE support for Maven tasks. All Maven tasks have to be invoked from the command line.

Describing the Java Community Process

Objectives

After completing this section, students should be able to:

- Describe the purpose of the Java Community Process (JCP).
- Describe the process by which the JCP defines, releases, and maintains Java Specification Requests (JSR).
- Explore specification and version numbers of APIs that constitute the Java EE 7 specification.

The Java Community Process (JCP)

The Java Community Process (JCP) is an open, participatory process to develop, maintain, and revise Java technology specifications. The JCP fosters evolution of the Java platform in cooperation with the international Java developer community.

Any individual or organization can join the JCP and participate in the standardization process. The JCP manages the specification of a large number of APIs using *Java Specification Requests (JSR)*.

Members are free to submit proposals for JSRs for any Java APIs. The proposal is reviewed by the JCP Executive Committee (EC), which consists of several senior Java community leaders elected by the JCP members. Once approved, a proposal is managed by a team of JCP members called as an *Expert Group (EG)*.

The expert group is responsible for defining, finalizing, and maintaining the specification of the API under the leadership of a Specification Lead (SL). When the JSR is ready for release, it is approved by the executive committee and becomes a JCP standard. Each JSR can be incrementally evolved and refined based on the needs of Java developers and current technology trends.

All individual APIs that constitute the Java EE 7 specification, itself managed in a separate JSR, have been developed under the JCP process and have individual JSRs managed by separate expert groups specializing in a particular technology domain.

The JCP, in collaboration with the expert groups, is also responsible for publishing a *Technology Compatibility Kit (TCK)*, a test suite that verifies whether an implementation conforms to the specification. For example, an application server is said to be "Java EE 7 compatible" only if it passes the Java EE 7 TCK fully and completely, without any errors or failures.

The Java EE 7 specification (JSR 342)

The Java EE 7 specification has been standardized and released under JSR 342. The Java EE 7 specification itself combines a number of APIs, each of which are standardized with their own unique JSR number. Different versions of the individual APIs have their own JSR numbers. For example, the EJB 3.1 specification is standardized under JSR 318, whereas EJB 3.0 was standardized under JSR 220, and the EJB 2.1 specification under JSR 153.

The Java EE 7 specification (JSR 342) is available at http://download.oracle.com/otndocs/jcp/java_ee-7-fr-eval-spec/index.html

For the detailed process followed by the JCP when creating and releasing JSRs, and a complete list of all JSRs and details on each, go through the links provided in the References at the end of this section.

The table below lists the APIs in the web and full profiles in Java EE 7 and their corresponding version and JSR numbers:

Profile	Specification	Version	JSR #
Web	Java API for WebSocket	1.0	356
	Java API for JSON Processing (JSON-P)	1.0	353
	Java Servlet API	3.1	340
	Java Server Faces (JSF)	2.2	344
	Expression Language (EL)	3.0	341
	Java Server Pages (JSP)	2.3	245
	JavaServer Pages Standard Tag Library (JSTL)	1.2	52
	Contexts and Dependency Injection (CDI)	1.1	346
	Dependency Injection for Java	1.0	330
	Bean Validation	1.1	349
	Managed Beans	1.0	316
	Enterprise JavaBeans (EJB) (EJB)	3.2	345
	Interceptors	1.2	318
	Java Persistence API (JPA)	2.1	338
	Common Annotations for the Java Platform	1.2	250
	Java Transaction API (JTA)	1.2	907
	Java API for RESTful Web Services (JAX-RS)	2.0	339
Full	Batch Applications for the Java Platform	1.0	352
	Concurrency Utilities for Java EE	1.0	236
	Java EE Connector Architecture (JCA)	1.7	322
	Java Message Service API (JMS)	2.0	343
	JavaMail API	1.5	919

Profile	Specification	Version	JSR #
	Java API for XML-Based Web Services (JAX-WS)	2.2	224
	Java Architecture for XML Binding (JAXB)	2.2	222
	Java API for XML Registries (JAXR)	1.0	93
	Java Authentication Service Provider Interface for Containers (JASPIC)	1.1	196

**Note**

The Java EE 7 *full profile* includes all of the APIs from the *web profile*.

**References****The Java Community Process (JCP)**

<https://jcp.org/en/home/index>

Java Specification Requests (JSR)

<https://jcp.org/en/jsr/overview>

The JCP Process

<https://jcp.org/en/procedures/overview>

► Quiz

Describing the Java Community Process (JCP)

Choose the correct answers to the following questions:

► 1. **Which of the following statements about the JCP is true?**

- a. Only organizations with revenues of 1 million USD or more are permitted to join the JCP.
- b. Only registered companies or organizations are allowed to join the JCP.
- c. Membership to the JCP is by invitation only. You cannot join the JCP as an individual or an organization without an invite.
- d. JCP membership is open to organizations, companies, and individuals.
- e. Only the JCP Executive Committee (EC) can propose a new Java Specification Request (JSR).

► 2. **Which of the following two statements about the JCP process are correct? (Choose two.)**

- a. Any individual member, company, or organization can propose a JSR.
- b. The Java EE 7 Technology Compatibility Kit (TCK) is just a test suite for verifying the implementation of the Java EE 7 specification. It is not mandatory for all the tests to pass for an implementation to be declared as compatible and certified.
- c. An implementation can be defined as certified and compatible if it passes 50% of the tests in the TCK.
- d. The JCP Executive Committee exclusively manages and oversees the development and evolution of each and every JSR.
- e. Expert Groups (EG) manage the development and evolution of JSRs under the leadership of a Specification Lead (SL).

► 3. **Which of the following two APIs are part of the Java EE 7 web profile? (Choose two.)**

- a. Java Connector Architecture (JCA)
- b. Java Authentication Service Provider Interface for Containers (JASPIC)
- c. Java Message Service API (JMS)
- d. Java API for RESTful Web Services (JAX-RS)
- e. Java Transaction API (JTA)

► 4. Which of the following two statements about the Java EE 7 full profile are correct?

(Choose two.)

- a. The Java Persistence API (JPA) is not a part of the Java EE 7 full profile.
- b. The Java API for XML-Based Web Services (JAX-WS) is not a part of the Java EE 7 full profile.
- c. The Java Message Service API (JMS) is a part of the Java EE 7 full profile.
- d. The full profile contains all the APIs in the web profile.
- e. The full profile contains only some APIs from the web profile.

► Solution

Describing the Java Community Process (JCP)

Choose the correct answers to the following questions:

► 1. **Which of the following statements about the JCP is true?**

- a. Only organizations with revenues of 1 million USD or more are permitted to join the JCP.
- b. Only registered companies or organizations are allowed to join the JCP.
- c. Membership to the JCP is by invitation only. You cannot join the JCP as an individual or an organization without an invite.
- d. JCP membership is open to organizations, companies, and individuals.
- e. Only the JCP Executive Committee (EC) can propose a new Java Specification Request (JSR).

► 2. **Which of the following two statements about the JCP process are correct? (Choose two.)**

- a. Any individual member, company, or organization can propose a JSR.
- b. The Java EE 7 Technology Compatibility Kit (TCK) is just a test suite for verifying the implementation of the Java EE 7 specification. It is not mandatory for all the tests to pass for an implementation to be declared as compatible and certified.
- c. An implementation can be defined as certified and compatible if it passes 50% of the tests in the TCK.
- d. The JCP Executive Committee exclusively manages and oversees the development and evolution of each and every JSR.
- e. Expert Groups (EG) manage the development and evolution of JSRs under the leadership of a Specification Lead (SL).

► 3. **Which of the following two APIs are part of the Java EE 7 web profile? (Choose two.)**

- a. Java Connector Architecture (JCA)
- b. Java Authentication Service Provider Interface for Containers (JASPIC)
- c. Java Message Service API (JMS)
- d. Java API for RESTful Web Services (JAX-RS)
- e. Java Transaction API (JTA)

► 4. Which of the following two statements about the Java EE 7 full profile are correct?

(Choose two.)

- a. The Java Persistence API (JPA) is not a part of the Java EE 7 full profile.
- b. The Java API for XML-Based Web Services (JAX-WS) is not a part of the Java EE 7 full profile.
- c. The Java Message Service API (JMS) is a part of the Java EE 7 full profile.
- d. The full profile contains all the APIs in the web profile.
- e. The full profile contains only some APIs from the web profile.

Describing Multi-tiered Application Architecture

Objectives

After completing this section, students should be able to explain multi-tiered applications and architectures.

Multi-tiered Application Architecture

Java EE applications are designed with a multi-tier architecture in mind. The application is split into components, each serving a specific purpose. Each component is arranged logically in a *tier*. Some of the tiers run on separate physical machines or servers. The application's business logic can run on application servers hosted in one data center, while the actual data for the database can be stored on a separate server.

The advantage of using tiered architectures is that as the application scales to handle more and more end users, each of the tiers can be independently scaled to handle the increased workload by adding more servers (a process known as "scale out"). There is also the added benefit that components across tiers can be independently upgraded without impacting other components.

In a classic web-based Java EE application architecture, there are four tiers:

- *Client Tier*: This is usually a browser for rendering the user interface on the end-user machines, or an applet embedded in a web page (increasingly rare).
- *Web Tier*: The web tier components run inside an application server and generate HTML or other markup that can be rendered or consumed by components in the client tier. This tier can also serve non-interactive clients such as other enterprise systems (both internal and external) via protocols such as Simple Object Access Protocol (SOAP) or Representational State Transfer (REST) web services.
- *Business Logic Tier*: The components in the business logic tier contain the core business logic for the application. These are usually a mix of Enterprise Java Beans (EJB), Plain Old Java Objects (POJO), Entity Beans, Message Driven Beans, and Data Access Objects (DAO), which interface with persistent storage systems such as RDBMS, LDAP, and others.
- *Enterprise Information Systems (EIS) Tier*: Many enterprise applications store and manipulate persistent data that is consumed by multiple systems and applications within an organization. Examples are relational database management systems (RDBMS), Lightweight Directory Access Protocol (LDAP) directory services, NoSQL databases, in-memory databases, mainframes, or other back-end systems that store and manage an organization's data securely.

Types of Multi-Tier Application Architectures

The Java EE specification is designed to accommodate many different types of multi-tier applications. Some of the most common ones are briefly highlighted below:

Web-centric architecture

This type of architecture is for simple applications with a browser-based front end and a simple back end powered by Servlets, Java Server Pages (JSP), or Java Server Faces (JSF). Features such as transactions, asynchronous messaging, and database access are not used.

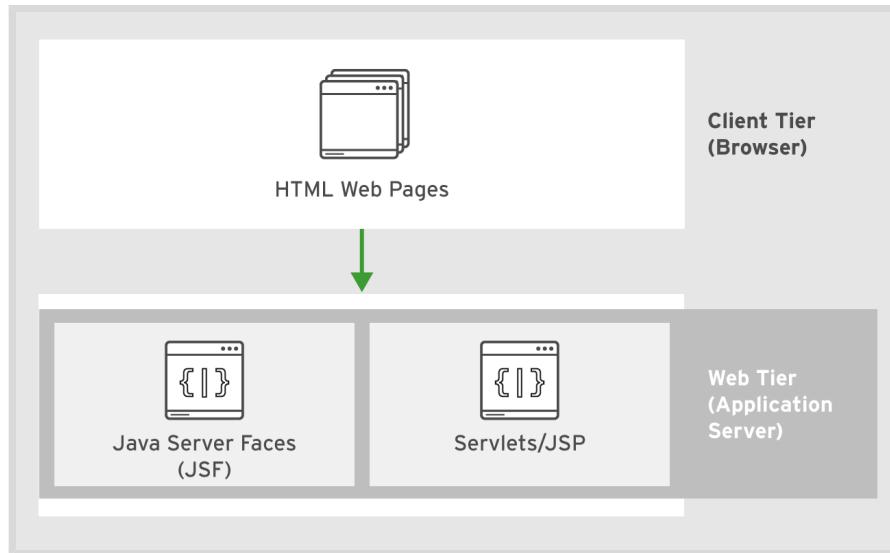


Figure 1.4: Simple Web-Centric architecture

Combined web and business logic component-based architecture

In this architecture, a browser in the client tier interfaces with a web tier consisting of Servlets, JSPs, or JSF pages, which are responsible for rendering the user interface, controlling page flow, and security. The core business logic is hosted in a separate business logic tier, which has Java EE components such as EJBs, Entity Beans (JPA), and Message Driven Beans (MDB). The business logic tier components integrate with enterprise information systems such as relational databases and back-office applications that expose an API for managing persistent data, and provide transactional capabilities for the application.

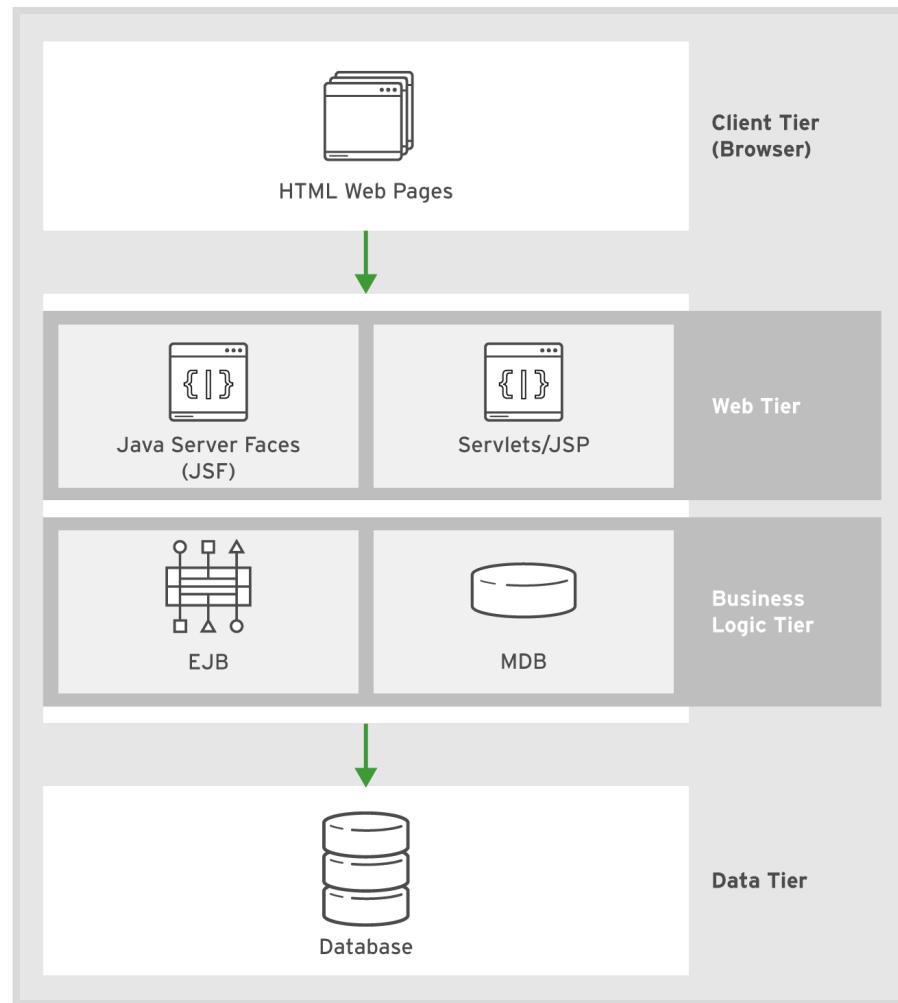


Figure 1.5: Combined web and business logic component-based architecture

Business-to-Business architecture (B2B)

In this type of architecture, the front end is usually not an interactive graphical user interface (GUI) that is accessed by end users, but an internal or external system that integrates with the application and exchanges data using a mutually understood standard protocol such as Remote Method Invocation (RMI), HTTP, Simple Object Access Protocol (SOAP), or Representational State Transfer (REST).

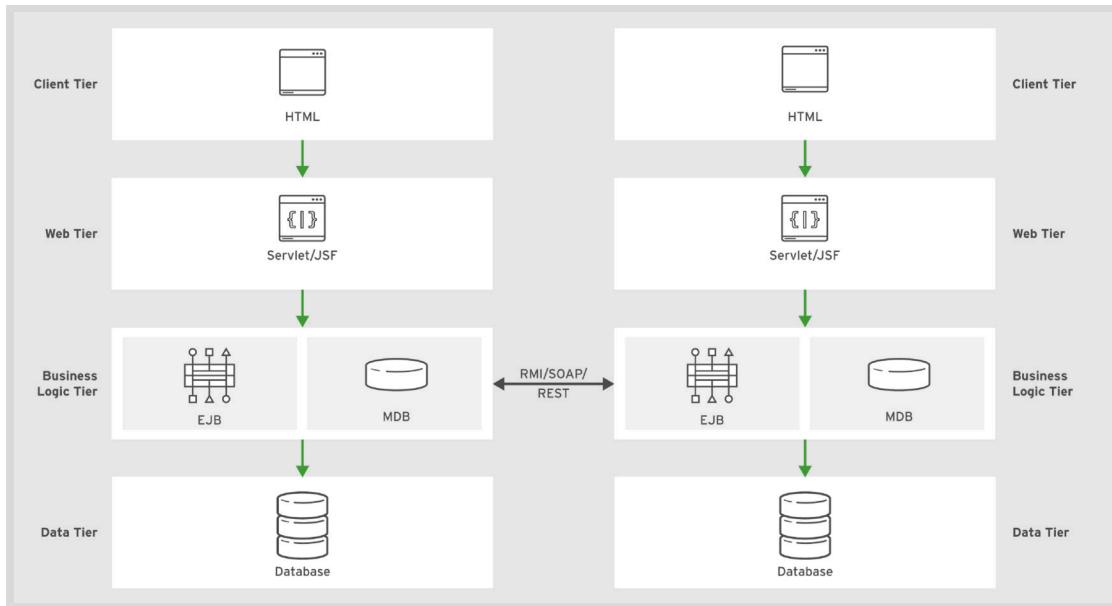


Figure 1.6: Business-to-Business architecture

Web service application architecture

Modern application architectures are often designed to be based on web services. In this architecture, the application provides an API that is accessed over an HTTP-based protocol such as SOAP or REST via a set of services (*endpoints*) corresponding to the business function of the application. These services are consumed by non-interactive applications (can be internal or third-party) or an interactive HTML/JavaScript front end using frameworks such as AngularJS, Backbone.js, React, and many more.

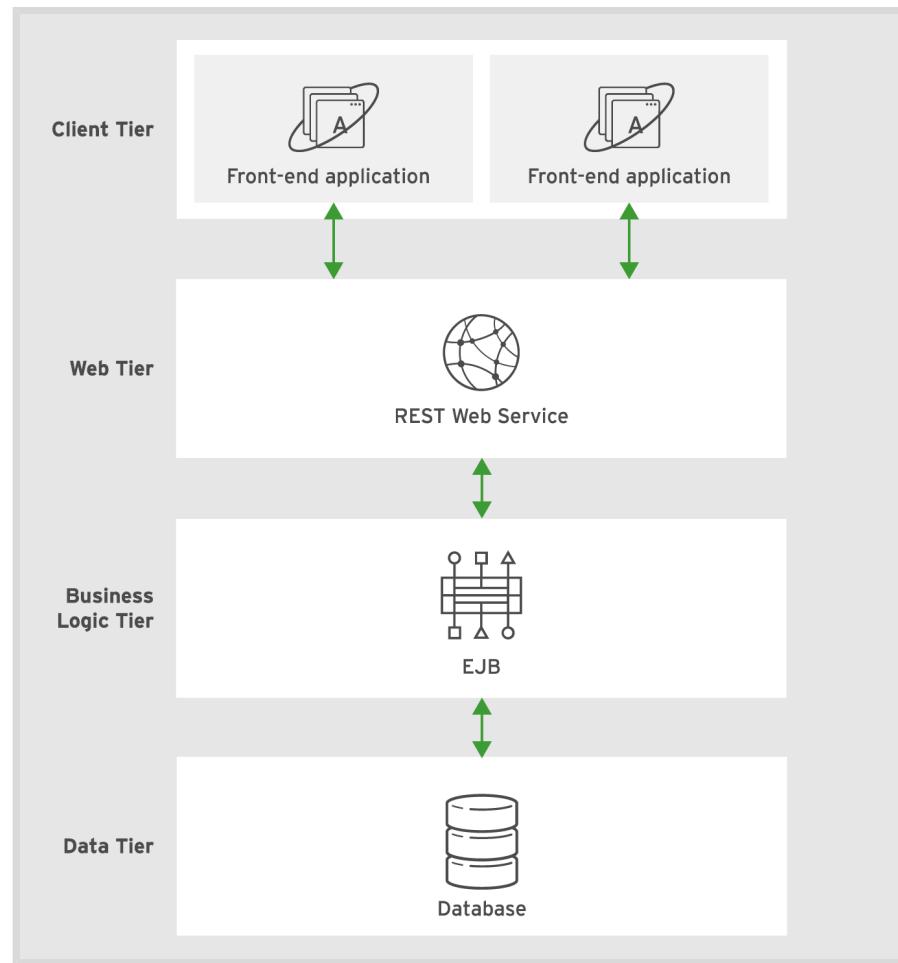


Figure 1.7: Simple web service application architecture

► Quiz

Multi-tiered Application Architecture

Choose the correct answers to the following questions:

- ▶ 1. **You have been asked to design a component that calculates discount rates for different products in an online shopping application. Which logical tier does this component best belong to?**
 - a. Client tier
 - b. Web tier
 - c. Business logic tier
 - d. Data or EIS Tier
 - e. None of the above

- ▶ 2. **Which of the following two applications are a good fit for a simple web-centric architecture? (Choose two.)**
 - a. A browser-based servlet application that prints the current time in three different time zones in the USA: Pacific (PST), Central (CST), and Eastern (EST).
 - b. An application that tracks the location of a fleet of cars using GPS.
 - c. An application that reads data from a large mainframe and then stores it in a relational database. The application also allows an external third-party system to access the data in the database using SOAP web services.
 - d. A health-check application that is deployed on an application server, which displays a status of "OK" (when accessed from a browser) if the server is up and running normally.
 - e. An application that provides weather information in cities around the world. The application accepts a city name as input (over a REST endpoint) and then provides current weather information and a 5-day forecast in XML format.

- ▶ 3. **Which of the following two statements about a Business-to-Business (B2B) architecture is correct? (Choose two.)**
 - a. B2B applications must be always web based and should have an interactive front end.
 - b. B2B applications must only support a single protocol for security reasons.
 - c. B2B applications always communicate using RMI.
 - d. B2B applications can communicate over RMI, SOAP, REST, or any mutually agreed-upon protocol.
 - e. B2B applications can support both interactive and non-interactive consumers and users.

- 4. Which of the following two statements about the combined web and business logic component architecture are correct? (Choose two.)
- a. Transactions are managed in the business logic tier (in EJBs).
 - b. Transactions must always be managed in the web layer.
 - c. Asynchronous messaging using Message Driven Beans (MDB) cannot be used.
 - d. Asynchronous messaging using Message Driven Beans (MDB) can be used.

► Solution

Multi-tiered Application Architecture

Choose the correct answers to the following questions:

- ▶ 1. **You have been asked to design a component that calculates discount rates for different products in an online shopping application. Which logical tier does this component best belong to?**
 - a. Client tier
 - b. Web tier
 - c. Business logic tier
 - d. Data or EIS Tier
 - e. None of the above

- ▶ 2. **Which of the following two applications are a good fit for a simple web-centric architecture? (Choose two.)**
 - a. A browser-based servlet application that prints the current time in three different time zones in the USA: Pacific (PST), Central (CST), and Eastern (EST).
 - b. An application that tracks the location of a fleet of cars using GPS.
 - c. An application that reads data from a large mainframe and then stores it in a relational database. The application also allows an external third-party system to access the data in the database using SOAP web services.
 - d. A health-check application that is deployed on an application server, which displays a status of "OK" (when accessed from a browser) if the server is up and running normally.
 - e. An application that provides weather information in cities around the world. The application accepts a city name as input (over a REST endpoint) and then provides current weather information and a 5-day forecast in XML format.

- ▶ 3. **Which of the following two statements about a Business-to-Business (B2B) architecture is correct? (Choose two.)**
 - a. B2B applications must be always web based and should have an interactive front end.
 - b. B2B applications must only support a single protocol for security reasons.
 - c. B2B applications always communicate using RMI.
 - d. B2B applications can communicate over RMI, SOAP, REST, or any mutually agreed-upon protocol.
 - e. B2B applications can support both interactive and non-interactive consumers and users.

- 4. Which of the following two statements about the combined web and business logic component architecture are correct? (Choose two.)
- a. Transactions are managed in the business logic tier (in EJBs).
 - b. Transactions must always be managed in the web layer.
 - c. Asynchronous messaging using Message Driven Beans (MDB) cannot be used.
 - d. Asynchronous messaging using Message Driven Beans (MDB) can be used.

Installing Java Development Tools

Objectives

After completing this section, students should be able to:

- Describe the JBoss Developer Studio editor features and installation process.
- Describe how to use Maven to manage application dependencies.

Red Hat JBoss Developer Studio (JBDS)

JBoss Developer Studio (JBDS) is an Integrated Development Environment (IDE) provided by Red Hat to simplify the development of Java EE applications. It is a set of integrated and well-tested plug-ins on top of the Eclipse™ platform. JBDS has the following built-in features:

- Plug-ins to simplify development of applications using Red Hat JBoss middleware.
- Unit testing plug-ins and wizards to do Test Driven Development (TDD).
- A visual debugger to help debug local and remote Java applications.
- Syntax highlighting and code completion for the most commonly used Java EE APIs, such as JPA, JSF, JSP, EL, and many more.
- Maven integration to simplify project builds, packaging, testing, and deployment.
- Unit adapters and plug-ins to work with JBoss EAP. You can control the life cycle (start, stop, restart, deployment, undeployment) of EAP without leaving the IDE.

Installing JBoss Developer Studio

JBDS has a platform independent JAR file installer and runs on Mac OS X, Windows, and Linux (GTK) using native UI widgets. A Java SDK is required. For JBDS 10 and later, a Java 1.8 SDK or newer is required. Start the install process by opening a terminal, and running the command:

```
[student@workstation todojse]$ java -jar  
devstudio-10.0.0.GA-standalone-installer.jar
```

The installer provides a series of wizards that prompts you for the location where you want to install the IDE, the JVM you want to use, if any JBoss EAP servers are installed, and if you want to add it to JBDS. Finally, it asks you to accept the end-user agreement before installing the IDE at the location you provided.

If you chose the option to automatically add shortcuts to the startup menu, you should see these shortcuts in your operating system's application menu. Double-click the JBDS icon to start the JBDS IDE.

**Note**

The JBDS installer makes a recording of the selections made during the graphical install, and stores it in a file named `InstallConfigRecord.xml` under the root installation directory. Using this, it is possible to replay an installation on multiple systems using the following command:

```
[student@workstation todojse]$ java -jar
devstudio-10.0.0.GA-standalone-installer.jar InstallConfigRecord.xml
```

Apache Maven

The current best practice for developing, testing, building, packaging, and deploying Java SE and Java EE applications is to use *Apache Maven*. Maven is a project management tool that uses a declarative approach (in an XML file called `pom.xml` at the root of the project folder) to specify how to build, package, execute (for Java SE applications), and deploy applications together with dependency information.

Maven has a small core and has a large number of plug-ins that extend the core functionality to provide features such as:

- Predefined build life cycles for end products, called *artifacts*, like WAR, EAR, and JAR.
- Built-in best practices such as source file locations and running unit tests for each build.
- Dependency management with automatic downloading of missing dependencies.
- Extensive plug-in collection including plug-ins specific to JBoss development and deployment.
- Project report generation including Javadocs, test coverage, and many more.

This section takes a look at the features and operational constructs of Maven that are used in this course. This all starts with the Maven project file, an XML document describing the artifact, its dependencies, project properties, and any plug-ins to be invoked in any of the available life-cycle steps. This file is always named `pom.xml`. The following is an abbreviated example of a project `pom.xml` file:

```
<?xml version="1.0"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd"
      xmlns="http://maven.apache.org/POM/4.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.redhat.training</groupId>1
  <artifactId>example</artifactId>2
  <version>0.0.1</version>3
  <packaging>war</packaging>4
  <name></name>
  <dependencies>5
    <dependency>
      <groupId>org.richfaces.ui</groupId>
      <artifactId>richfaces-components-ui</artifactId>
      <version>4.0.0.Final</version>
    </dependency>
  ...

```

```

</dependencies>
<build>
  <plugins>⑥
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>2.3.2</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
  </plugins>
  ...
</build>
</project>

```

- ①** The `group-id` is like a Java package.
- ②** The `artifact-id` is a project name.
- ③** The `version` of the project.
- ④** The `packaging` defines how the project will be packaged. In this case it is a `war` type.
- ⑤** The `dependency` describes the resources a project depends on. These resources are required to build and run the project correctly. Maven downloads and links the dependencies from the specified repositories.
- ⑥** The `plugins` for the project.

A benefit of using Maven is the automatic handling of source code compilation and resource inclusion in the artifact. Maven creates a standard project structure. The following directory naming conventions are mandatory:

Maven Directory Structures

Asset	Directory	Outcome
Java Source Code	<code>src/main/java</code>	The directory contains Java classes included in <code>WEB-INF/classes</code> for a WAR or root of a JAR.
Configuration Files	<code>src/main/resources</code>	The directory contains configuration files included in <code>WEB-INF/classes</code> for a WAR or root of JAR.
Java Test Code	<code>src/test/java</code>	The directory contains the test source code.
Test Configuration Files	<code>src/test/resources</code>	The directory contains the test resources.

When you name dependencies in the `pom.xml` file, you can give them a scope. These scopes control where the dependency is used in the build life cycle and whether they are included in the artifact. The following scopes are the most common:

Maven Dependency Scopes

Scope	Outcome
compile	Compile is the default scope if no other scope is specified and is required to resolve the <code>import</code> statements.
test	Test is required to compile and run the unit tests. It is not included in the artifact.
runtime	The runtime dependency is not required for compilation. It is used for any executions and is included in the artifact.
provided	Provided scope is like compile and container provides the dependency at runtime. It is used during build and test.

Maven is integrated in JBDS, but you may want to invoke it from the command line. Here are some common commands:

- `mvn package` - Compiles, tests, and builds the artifact.
- `mvn package -Dmaven.test.skip=true` - Builds the artifact and skip all tests.
- `mvn jboss-as:deploy` - To deploy the artifact to the instance running at `$JBOSS_HOME` (assumes plug-in configured in `pom.xml`).
- `mvn install` - This is like the package but installs the artifact in your local Maven repository for use in other projects as a dependency.



Note

IDEs such as JBoss Developer Studio are aware of Maven projects and you can run Maven tasks directly from within the IDE without requiring the use of the command line.

Throughout this course you will be incrementally developing a web-based To Do List Application that is deployed on a JBoss EAP 7 application server and uses several APIs from the Java EE 7 specification. The application stores the data in a MySQL database. You will make use of Maven and JBDS extensively in this course to manage the application packaging and deployment.

To build, package, and run a standalone application that uses only the Java SE API, such as the command-line based To Do List Application using Maven, you will run the following commands:

```
[student@workstation todojse]$ mvn clean package  
[student@workstation todojse]$ java -jar target/todojse-1.0.jar
```

The `mvn clean package` command builds the application as an executable JAR file and the `java -jar` command executes it.

In contrast, the web-based To Do List Application is built and deployed to EAP by using the following command:

```
[student@workstation todojse]$ mvn clean package wildfly:deploy
```

The above command deletes the old WAR file, compiles the code, and builds a WAR file that is deployed to a running instance of EAP. If an older version of the application is already deployed, the old version is undeployed and the new version is deployed without restarting the application server. This process is called *hot deployment* and is used extensively during development and testing, as well as in production rollouts.



References

Eclipse

<https://eclipse.org>

Apache Maven

<https://maven.apache.org>

► Workshop

Running the To Do List Application

Outcomes

You should be able to import the To Do List Application command-line project into JBoss Developer Studio and run it using Maven.

Before You Begin

Open a terminal window on the **workstation** VM and run the following command to download the lab files required for this workshop.

```
[student@workstation ~]$ lab todojse setup
```

- ▶ 1. Import the todojse project into JBoss Developer Studio IDE (JBDS).
 - 1.1. Double-click the JBDS icon on the **workstation** VM desktop.
 - 1.2. Select a workspace folder.

In the **Eclipse Launcher** window, enter **/home/student/JB183/workspace** in the **Workspace** field, check the **Use this as the default and do not ask again** checkbox, and then click **Launch**.

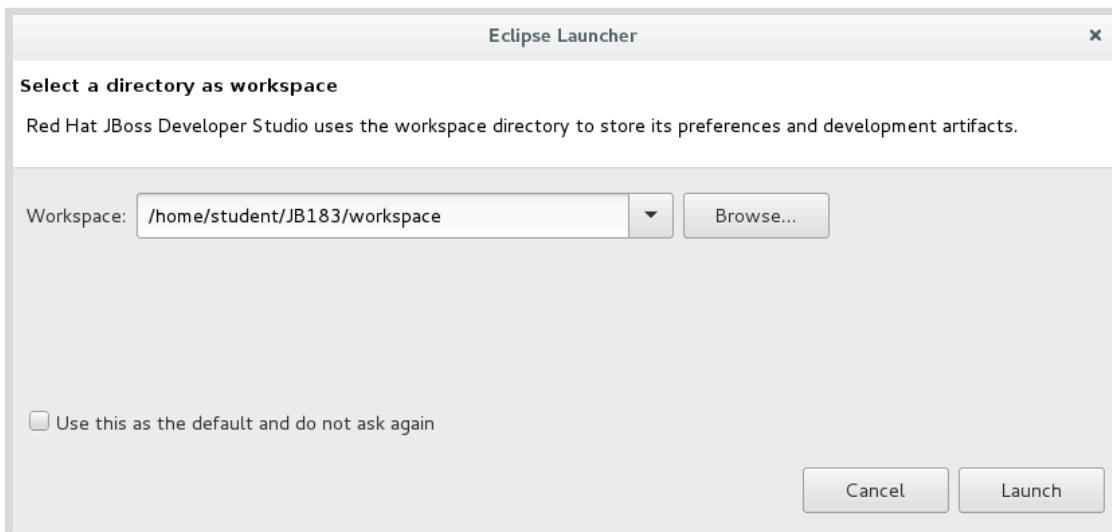


Figure 1.8: Select JBDS workspace



Note

Select **No** to dismiss the Red Hat JBoss Developer Studio Usage dialog box.

- 1.3. In the JBDS menu, click **File > Import** to open the **Import** wizard.
On the **Select** page, select **Maven > Existing Maven Projects**, and then click **Next**.

- 1.4. In the **Maven projects** page, click **Browse** to open the **Select Root Folder** window. Navigate to the `/home/student/JB183/labs/` directory. Select the `todojse` folder, and then click **OK**.
On the **Maven projects** page, click **Finish**.
- 1.5. Watch the JBDS status bar (lower-right corner) to monitor the progress of the import operation. It may take a few minutes to download all of the required dependencies.

► 2. Explore the Maven `pom.xml` file.

- 2.1. Expand the `todojse` item in the **Project Explorer** pane on the left, and then double-click the `pom.xml` file.

The **Overview** tab is seen in the main editor window, showing a high-level view of the project. The **Group Id**, **Artifact Id** and the **Version** (commonly abbreviated as the GAV coordinates of a project or module) is shown in this tab.

- 2.2. Click the **Dependencies** tab to view the dependencies (libraries, frameworks, and modules that this project depends on) for the project. In this case, we have no dependencies on any external libraries and only utilize the Java Standard Library.

- 2.3. Click the **pom.xml** tab to view the full text of the `pom.xml` file.

Briefly review the GAV details for this project:

```
<groupId>com.redhat.training</groupId>
<artifactId>todojse</artifactId>
<version>1.0</version>
```

The packaging format for this project as `jar`. Maven ensures that when the project is built, it will result in a JAR file with appropriate **MANIFEST** entries containing metadata about the jar file.

```
<packaging>jar</packaging>
```

The project is compatible with JDK 1.8.

```
<!-- maven-compiler-plugin -->
<maven.compiler.target>1.8</maven.compiler.target>
<maven.compiler.source>1.8</maven.compiler.source>
```

Maven is extensible by using a large number of *plug-ins*. You can control different aspects of how your project is built, packaged, tested, and deployed by declaring appropriate plug-ins.

In this project, you are using the `exec-maven-plugin` to run the main class in the project from the command line or from within the JBoss Developer Studio. The main method, which serves as the entry point for the application when it is run, is declared as the `com.redhat.training.TestTodoMap` class.

```
<artifactId>exec-maven-plugin</artifactId>
<version>1.5.0</version>
<executions>
  <execution>
    <goals>
      <goal>java</goal>
```

```

</goals>
</execution>
</executions>
<configuration>
  <mainClass>com.redhat.training.TestTodoMap</mainClass>
</configuration>

```

You also use the `maven-assembly-plugin` to build a platform-independent executable JAR file, which can be run using the `java -jar` command. Although, this project does not use any external dependencies, projects with a large number of dependent JAR files can be packaged as a single large *fat jar* that can be directly executed without explicitly adding all the dependent JAR files to the CLASSPATH.

```

<artifactId>maven-assembly-plugin</artifactId>
<version>2.6</version>
<executions>
  <execution>
    <id>package-jar-with-dependencies</id>
    <phase>package</phase>
    <goals>
      <goal>single</goal>
    </goals>
  </execution>
</executions>

```

► 3. Explore the application source code.

- 3.1. Navigate to `src/main/java/com/redhat/training` in the **Project Explorer** and double-click the `TestTodoMap.java` class to view the source code in the main editor window.
- 3.2. The todojse application is a command-line application with no graphical user interfaces. The `main` method invokes the `executeMenu()` function, which displays a menu to the user with a number of options to manage To Do list.

The `TodoMap.java` class contains the main business logic for this application. This class stores and manages a Map of `TodoItem` objects. The `TodoItem` class is a simple Java Bean class that encapsulates the attributes of a To Do List; namely, an `item` field, which contains the task description, and a `status` field that indicates if the task is pending or complete.

The `Status.java` file declares an enum with the two options for the status of an item, either `PENDING` or `COMPLETED`.

- 3.3. Briefly review the source code of the `addTodo()`, `printTodo()`, `completeTodo()`, `deleteTodo()` and `findItemTodo()` methods in the `TodoMap` class to understand how tasks are created, listed, marked as completed, deleted, and found respectively.

These methods are invoked from the switch, or case, statement in the main runnable class, depending on which menu option is selected by the user. If the user selects Q, then the application exits.



Note

The todojse application does not persist any data from the program. The To Do list task items are stored in a Map object in memory and the data is lost when the program exits.

► **4.** Build and run the todojse from the command line using Maven.

- 4.1. Before running the application from within the JBDS IDE, build and run the application from the command line using Maven.

Open a new terminal window and navigate to the /home/student/JB183/labs/todojse folder:

```
[student@workstation ~]$ cd /home/student/JB183/labs/todojse
```

You can now build and package the application as a JAR file using Maven's package goal.

- 4.2. Build the application by running the following command:

```
[student@workstation todojse]$ mvn clean package
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building todojse 1.0
[INFO] -----
[INFO]
[INFO]
...
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ todojse ---
[INFO] Building jar: /home/student/JB183/labs/todojse/target/todojse-1.0.jar
...
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 09:48 mins
[INFO] Finished at: 2017-09-20T08:13:03+05:30
[INFO] Final Memory: 22M/303M
```

Verify that you can see a `BUILD SUCCESS` message from Maven and the `todojse-1.0.jar` is built successfully and copied to the `/home/student/JB183/labs/todojse/target` folder.

- 4.3. Run the application using the Maven exec plug-in:

```
[student@workstation todojse]$ mvn exec:java
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building todojse 1.0
[INFO] -----
[INFO]
[INFO] --- exec-maven-plugin:1.5.0:java (default-cli) @ todojse ---

[N]ew | [C]omplete | [R]ead | [D]elete | [L]ist | [Q]uit:
```

- 4.4. Explore the application functionality by creating, completing, reading, listing, and deleting a few to do items. Enter Q to quit the application.

► 5. Run the todojse as an executable JAR file.

- 5.1. The `mvn clean package` command you ran earlier uses the Maven assembly plug-in to build a stand-alone executable JAR file.

Run the application using the following command:

```
[student@workstation todojse]$ java -jar target/todojse-1.0.jar
```

- 5.2. Verify that the application is launched and that the main menu is displayed.

► 6. Build and run the todojse from within JBDS.

- 6.1. You can build, package, and run the application from within JBDS by using the built-in Maven plug-in in the IDE.

The JBDS Maven plug-in comes with a set of prepackaged *Run Configurations* for cleaning and building projects. However, you will create a custom *Run Configuration* and use it to build, package, and run the project.

- 6.2. Right-click on the `todojse` project in the **Project Explorer** and click **Run As > Run Configurations** to launch the **Run Configurations** window.

Scroll down the list of options in the left panel and then select the **Maven Build** option:

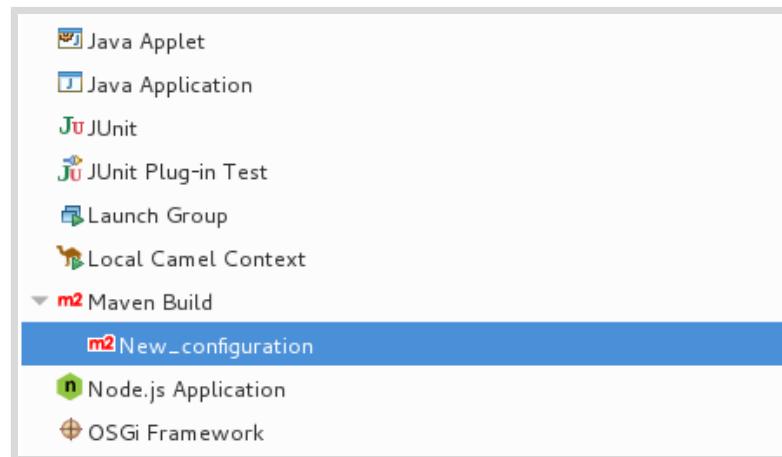


Figure 1.9: Maven run configuration

- 6.3. In the upper-left menu in the **Run Configurations** window, click **New Launch Configuration** to create a new Maven launch configuration:

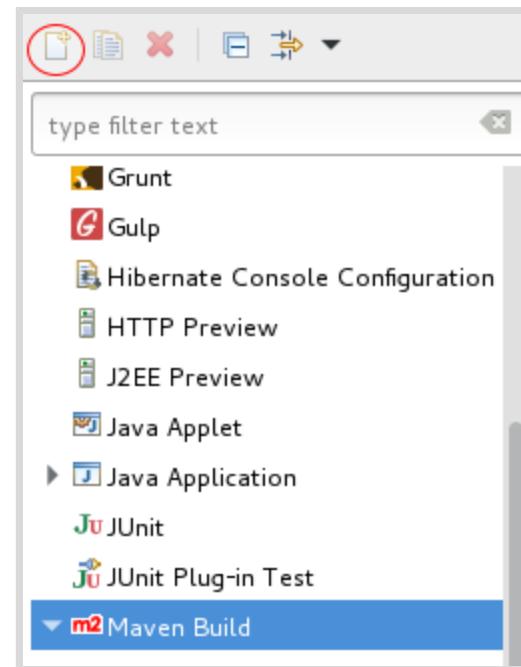


Figure 1.10: New run configuration

6.4. In the new run configuration window, add the following details:

- Name: **maven package and exec**
- Base Directory: Click Workspace and then select the todojse project and click OK.
- Goals: **clean package exec:java**

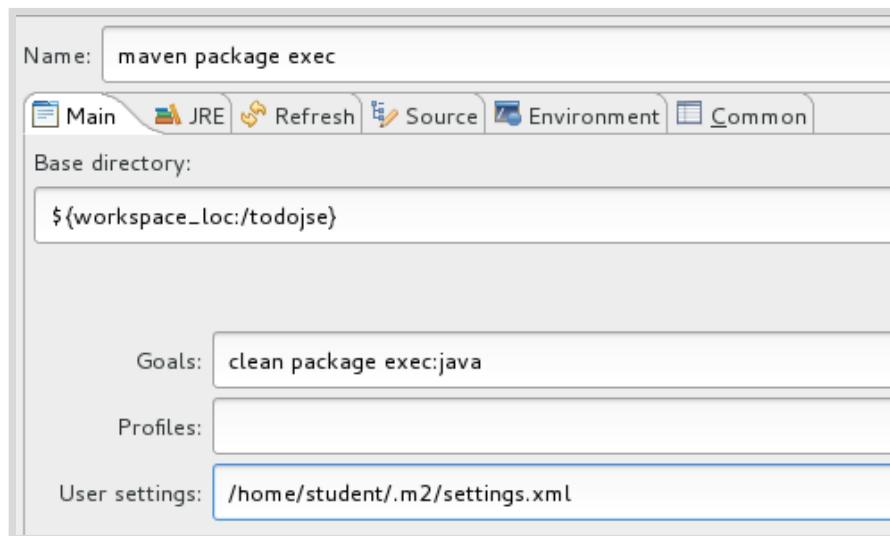


Figure 1.11: Run configuration details for Maven

Leave all other fields at default values and click **Apply**.

6.5. Click **Run** at the bottom of the Run Configurations window.

The JBDS Maven plug-in should now launch and build, package, and execute the application. Observe the **Console** tab at the bottom to watch the build process and verify that the application is executed and the main menu is displayed.

- 6.6. Exit the application by entering **Q** in the To Do List Application main menu.
- 6.7. Right click on the `todojse` project in the **Project Explorer**, and select **Close Project** to close this project.

This concludes the guided exercise.

Summary

In this chapter, you learned:

- *Enterprise applications* are characterized by their ability to handle transactional workloads, multi-component integration, security, distributed architectures, and scalability.
- *Java Enterprise Edition (Java EE)* is a specification for developing enterprise applications using Java. It is a platform-independent standard that is developed under the aegis of the Java Community Process (JCP). A software system that implements the Java EE specification is called an *application server*.
- The Java SE API provides a rich set of modular, reusable components for implementing Java applications. Java EE is built on top of Java SE and provides a set of APIs that are focused on developing enterprise applications.
- Java EE applications are designed to be multi-tiered and can accommodate a variety of architectures depending on the use case.
- *Red Hat JBoss Developer Studio* is an Eclipse™ based IDE provided by Red Hat that contains a set of integrated plug-ins and tools to simplify development of Java EE enterprise applications. It supports many application servers and you can manage the life cycle of the application server from within the IDE itself.
- *Apache Maven* is the preferred tool for building, packaging, and deploying Java SE and Java EE applications. JBDS has built-in support for Maven. Projects can be built, tested, packaged, and deployed to application servers using Maven plug-ins.

Chapter 2

Packaging and Deploying a Java EE Application

Overview

Goal

Describe the architecture of a Java EE application server, package an application, and deploy the application to an EAP server.

Objectives

- Identify the key features of application servers and describe the Java EE server architecture.
- List the most common JNDI resource types and their typical naming convention.
- Package a simple Java EE application and deploy it to JBoss EAP using Maven.

Sections

- Describing an Application Server (and Quiz)
- Identifying JNDI Resources (and Guided Exercise)
- Packaging and Deploying a Java EE Application (and Guided Exercise)

Lab

- Packaging and Deploying a Java EE Application

Describing an Application Server

Objectives

After completing this section, students should be able to:

- Identify key features of application servers and describe the Java EE server architecture.
- Identify various types of containers and server profiles.

Application Servers

An *application server* is a software component that provides the necessary runtime environment and infrastructure to host and manage Java EE enterprise applications. The application server provides features such as concurrency, distributed component architecture, portability to multiple platforms, transaction management, web services, object relational mapping for databases (ORM), asynchronous messaging, and security for enterprise applications.

In a Java SE application, these features must be implemented manually by the developer, which is time consuming and difficult to implement correctly.

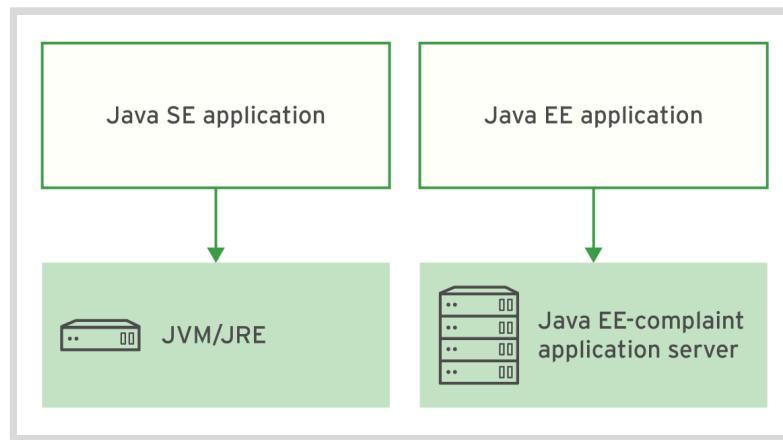


Figure 2.1: Java SE versus Java EE applications

JBoss Enterprise Application Platform (EAP)

Red Hat JBoss Enterprise Application Platform 7, JBoss EAP 7, or simply EAP, is an *application server* to host and manage Java EE applications.

EAP 7 is built on open standards, based on the **Wildfly** open source software, and provides the following features:

- A reliable, standards compliant, light-weight, and supported infrastructure for deploying applications.
- A modular structure that allows users to enable services only when they are required. This improves performance and security, and reduces start and restart times.
- A web-based management console and management command-line interface (CLI) to configure the server and provide the ability to script and automate tasks.

Chapter 2 | Packaging and Deploying a Java EE Application

- It is certified for both Java EE 7 **full**, and **web** profiles.
- A centralized management of multiple server instances and physical hosts.
- Preconfigured options for features such as high-availability clustering, messaging, and distributed caching are also provided.

EAP 7 makes developing enterprise applications easier because it provides Java EE APIs for accessing databases, authentication, and messaging. Common application functionality is also supported by Java EE APIs and frameworks, which are provided by EAP, for developing web user interfaces, exposing web services, implementing cryptography, and other features. JBoss EAP also makes management easier by providing runtime metrics, clustering services, and automation.

EAP has a modular architecture with a simple core infrastructure that controls the basic application server life cycle and provides management capabilities. The core infrastructure is responsible for loading and unloading *modules*. Modules implement the bulk of the Java EE 7 APIs. Each Java EE component API module is implemented as a *subsystem*, which can be configured, added, or removed as required through EAP's configuration file or management interface. For example, to configure access to a database in EAP, configure the database connection details in the **datasources** subsystem.

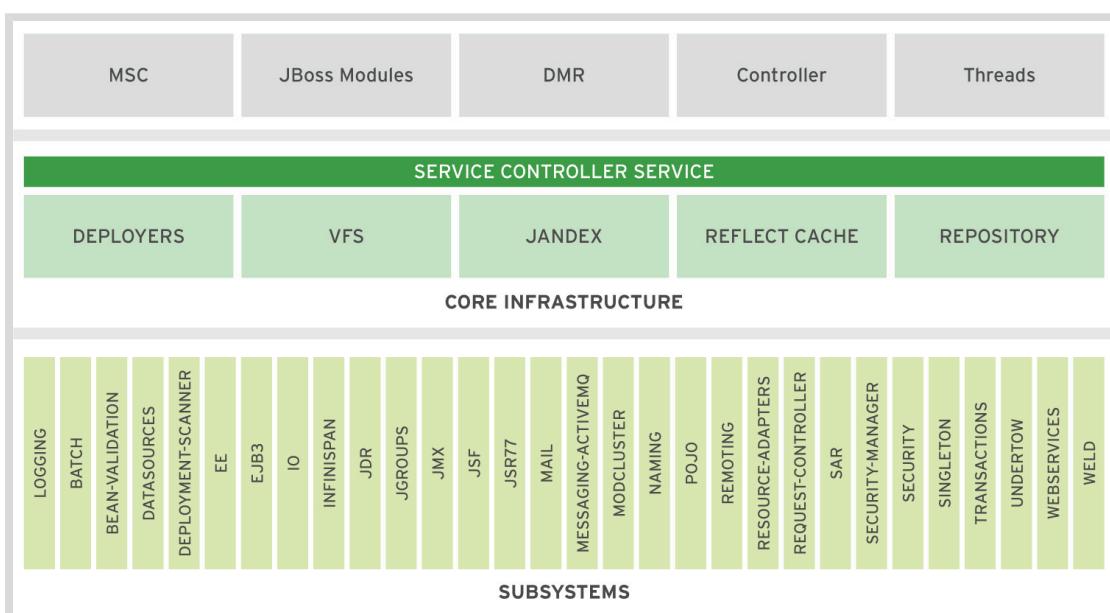


Figure 2.2: EAP 7 architecture

An important concept of the EAP architecture is the concept of a *module*. A module provides code (Java Classes) to be used by EAP services or by applications.

Modules are loaded into an isolated **Class loader**, and can only see classes from other modules when explicitly requested. This means a module can be implemented without any concerns about potential conflicts with the implementation of other modules. All code running in EAP, including the code provided by the core, runs inside modules. This includes application code, which means that applications are isolated from each other and from EAP services.

This modular architecture allows for a very fine-grained control of code visibility. An application can see a module that exposes a particular version of an API, while another application may see a second module that exposes a different version of the same API.

An application developer can control this visibility manually and it can be very useful in some scenarios. But for most common cases, EAP 7 automatically decides which modules to expose to an application, based on its use of Java EE APIs.

Containers

A *container* is a logical component within an application server that provides a runtime context for applications deployed on the application server. A container acts as an interface between the application components and the low-level infrastructure services provided by the application server.

There are different containers for different types of components in an application. Application components are *deployed* to containers and made available to other deployments. Deployment is based on the deployment descriptors (XML configuration files that are packaged alongside the code) or code-level annotations that indicate how the components should be deployed and configured.

There are two main types of containers within a Java EE application server:

- **Web containers:** Deploy and configure web components such as Servlets, JSP, JSF, and other web-related assets.
- **EJB containers:** Deploy and configure EJB, JPA, and JMS-related components. These types of deployments are described in detail in later chapters.

Containers are responsible for security, transactions, JNDI lookups, and remote connectivity and more. Containers can also manage runtime services, such as EJB and web component life cycles, data source pooling, data persistence, and JMS messaging. For example, the Java EE specification allows you to declaratively configure security so that only authorized users can invoke functionality provided by an application component. This restriction is configured using either XML deployment descriptors or annotations in code. This metadata is read by the container at deployment time and it configures the component accordingly.

Java EE 7 Profiles

A *profile* in the context of a Java EE application server is a set of component APIs that target a specific application type. Profiles are a new concept introduced in Java EE 6. There are currently two profiles defined in Java EE 7 and the JBoss EAP application server fully supports both profiles:

- **Full Profile:** Contains all Java EE technologies, including all APIs in the web profile as well as others.
- **Web Profile:** Contains a full stack of Java EE APIs for developing dynamic web applications.

There are over 30 different technologies that comprise the full profile of Java EE. Each of these technologies has their own JSR specification and version number. Combined, they provide an impressive list of capabilities that allow Java EE applications to connect to databases, publish and consume web services, serve up web applications, perform transactions, implement security policies, and connect to a multitude of external resources for tasks such as messaging, naming, sending emails, and communicating with non-Java applications.

The web profile contains the web-based technologies of Java EE that are commonly used by web developers, such as Servlets, Java Server Pages, Java Server Faces, CDI, JPA, JAX-RS, WebSockets, and a limited version of Enterprise Java Beans (EJBs) known as EJB Lite. Many of these technologies are described in detail throughout this course.



References

Further information is available in the Introduction to JBoss EAP chapter of the *Development Guide* for Red Hat JBoss EAP 7.0:
<https://access.redhat.com/documentation/en/red-hat-jboss-enterprise-application-platform/>

► Quiz

Describing an Application Server

Choose the correct answers to the following questions:

► 1. **Which of the following statements about Java SE and Java EE applications is true?**

- a. Java EE applications are hosted and managed by an application server.
- b. Java SE applications cannot connect to a database and perform transactions.
- c. Java SE applications are always single-threaded. Only Java EE applications are multi-threaded.
- d. Java EE applications cannot perform asynchronous messaging.
- e. In Java EE applications, multi-threading and concurrency has to be manually implemented by the developer.

► 2. **Which of the following three statements about JBoss EAP are correct? (Choose three.)**

- a. EAP is based on the Tomcat open source web server.
- b. EAP is based on the Wildfly open source application server.
- c. All the Java EE APIs are bundled as modular components into EAP 7. All the modules are enabled by default, and you cannot add or remove any modules.
- d. All the Java EE APIs are bundled as modular components into EAP 7. The modules can be enabled, only when required.
- e. In EAP 7, high-availability clustering, messaging, and distributed caching are not available by default. You have to use third-party products to enable these features.
- f. EAP 7 supports both the Java EE 7 web and full profiles.

► 3. **Which of the following two statements about the EAP 7 architecture are correct? (Choose two.)**

- a. The concept of modules applies only to the EAP-provided services. Applications cannot be run as modules.
- b. Both user-developed applications and EAP-provided services can be run as modules.
- c. Modules have global visibility scope; that is, all modules can access classes from other modules in EAP implicitly.
- d. EAP has exclusive control over visibility. Classes from other modules have to be explicitly requested.

► **4. Which of the following statements about containers in an application server is true?**

- a. There must only be one container in an application server. Multiple containers within a single application server are not allowed.
- b. The containers can only read XML deployment descriptors in deployments. Code-level annotations are not supported.
- c. Application components can declaratively configure security in XML deployment descriptors or code-level annotations.
- d. A web container can deploy EJB, JMS, and JPA components.
- e. The EJB container must be run separately, outside of the application server. The application server only supports running the web container inside it.

► **5. Which of the following two statements about the Java EE profiles are correct? (Choose two.)**

- a. A profile is a collection of APIs focused on specific application types.
- b. The concept of a profile was introduced in Java EE 7.
- c. The Java EE 7 specification defines three profiles: *web*, *ejb*, and *full*.
- d. The Java EE 7 specification defines four profiles: *web*, *ejb*, *jms*, and *full*.
- e. The Java EE 7 specification defines two profiles: *web* and *full*.

► **6. Which of the following two statements about the web profile are correct? (Choose two.)**

- a. The *web* profile contains all APIs in the *full* profile as well as other APIs focused on web technologies.
- b. JBoss EAP does not support the *web* profile.
- c. The *web* profile has support for JMS Message Driven Beans (MDB).
- d. EJB Lite is part of the *web* profile.
- e. CDI is part of the *web* profile.

► Solution

Describing an Application Server

Choose the correct answers to the following questions:

► 1. **Which of the following statements about Java SE and Java EE applications is true?**

- a. Java EE applications are hosted and managed by an application server.
- b. Java SE applications cannot connect to a database and perform transactions.
- c. Java SE applications are always single-threaded. Only Java EE applications are multi-threaded.
- d. Java EE applications cannot perform asynchronous messaging.
- e. In Java EE applications, multi-threading and concurrency has to be manually implemented by the developer.

► 2. **Which of the following three statements about JBoss EAP are correct? (Choose three.)**

- a. EAP is based on the Tomcat open source web server.
- b. EAP is based on the Wildfly open source application server.
- c. All the Java EE APIs are bundled as modular components into EAP 7. All the modules are enabled by default, and you cannot add or remove any modules.
- d. All the Java EE APIs are bundled as modular components into EAP 7. The modules can be enabled, only when required.
- e. In EAP 7, high-availability clustering, messaging, and distributed caching are not available by default. You have to use third-party products to enable these features.
- f. EAP 7 supports both the Java EE 7 *web* and *full* profiles.

► 3. **Which of the following two statements about the EAP 7 architecture are correct? (Choose two.)**

- a. The concept of modules applies only to the EAP-provided services. Applications cannot be run as modules.
- b. Both user-developed applications and EAP-provided services can be run as modules.
- c. Modules have global visibility scope; that is, all modules can access classes from other modules in EAP implicitly.
- d. EAP has exclusive control over visibility. Classes from other modules have to be explicitly requested.

► **4. Which of the following statements about containers in an application server is true?**

- a. There must only be one container in an application server. Multiple containers within a single application server are not allowed.
- b. The containers can only read XML deployment descriptors in deployments. Code-level annotations are not supported.
- c. Application components can declaratively configure security in XML deployment descriptors or code-level annotations.
- d. A web container can deploy EJB, JMS, and JPA components.
- e. The EJB container must be run separately, outside of the application server. The application server only supports running the web container inside it.

► **5. Which of the following two statements about the Java EE profiles are correct? (Choose two.)**

- a. A profile is a collection of APIs focused on specific application types.
- b. The concept of a profile was introduced in Java EE 7.
- c. The Java EE 7 specification defines three profiles: *web*, *ejb*, and *full*.
- d. The Java EE 7 specification defines four profiles: *web*, *ejb*, *jms*, and *full*.
- e. The Java EE 7 specification defines two profiles: *web* and *full*.

► **6. Which of the following two statements about the web profile are correct? (Choose two.)**

- a. The *web* profile contains all APIs in the *full* profile as well as other APIs focused on web technologies.
- b. JBoss EAP does not support the *web* profile.
- c. The *web* profile has support for JMS Message Driven Beans (MDB).
- d. EJB Lite is part of the *web* profile.
- e. CDI is part of the *web* profile.

Identifying JNDI Resources

Objective

After completing this section, students should be able to list the most common JNDI resource types and their typical naming convention.

JNDI Resources

In a distributed multi-tier application running different components across multiple servers, components need to communicate with each other. For example, a Java client might invoke methods on an EJB that is deployed on a separate machine, and the EJB component communicates with a database to retrieve data. The *Java Naming and Directory Interface (JNDI)* is a Java API for a directory service (for looking up resources) that allows components to discover and look up objects via a logical *name*.

A *resource* is a logical object that can be looked up and used by components in a Java EE application. Each resource is identified by a unique name, called the *JNDI name* or a *JNDI resource binding*. For example, the JNDI name of the Java Database Connectivity (JDBC) data source that is provided by default (pointing to an embedded H2 database) in JBoss EAP is `java:jboss/datasources/ExampleDS`.

JNDI resources are not just restricted to JDBC data sources. Multiple types of resources are configurable, such as JMS ConnectionFactory objects, messaging queues and topics, email servers, thread pools, and others.

Each of the different JNDI bindings are organized under a logical *namespace*, in a tree-like hierarchy, usually called the *JNDI tree*. The following are some of the most common namespaces in the JBoss EAP application server:

- JDBC data sources are registered under the `java:jboss/datasources/*` namespace.
- JMS-related resources are registered under the `java:jboss/jms/*` namespace (JMS Queues under `java:jboss/jms/queue/*` and Topics under `java:jboss/jms/topic/*`).
- Email-related resources are registered under the `java:jboss/mail/*` namespace.
- Threading and concurrency-related resources are registered under the `java:jboss/ee/concurrency/*` namespace.

Resource Injection Using CDI

Java EE 7 provides Contexts and Dependency Injection (CDI) to enable components to obtain references to other component objects as well as application server resources without manually instantiating the server resources or component objects. This enables a loosely-coupled architecture where the client does not need to be aware of all the low-level implementation details of the invoked object.

After configuring the required JNDI resource bindings at the application server level, you can **inject** the resources into applications that require the resource by using an `@Resource` annotation. The application server instantiates the resource at runtime and provides a reference to the resource.

For example, assume you have configured a JDBC data source binding such as the following in an EAP configuration file:

```
<subsystem xmlns="urn:jboss:domain:datasources:4.0">
<datasources>
    <datasource jndi-name="java:jboss/datasources/ExampleDS" pool-name="ExampleDS"
        enabled="true" use-java-context="true">
        <connection-url>jdbc:h2:mem:test;DB_CLOSE_DELAY=-1;DB_CLOSE_ON_EXIT=FALSE</connection-url>
        <driver>h2</driver>
        <security>
            <user-name>sa</user-name>
            <password>sa</password>
        </security>
    </datasource>
</datasources>
...
```

You can now inject the `java:jboss/datasources/ExampleDS` data source into the application as follows:

```
public class TestDS {
    @Resource(name="java:jboss/datasources/ExampleDS")
    private javax.sql.DataSource ds;
    // Use the DataSource reference to create a Connection etc..
}
```

Similarly, if you have set up a JMS Queue resource like the following in EAP:

```
<jms-queue name="helloWorldQueue" entries="java:jboss/jms/queue/helloWorldQueue"/>
```

You can now send messages to this queue by injecting the resource to a JMS client class:

```
@Resource(mappedName = "java:jboss/jms/queue/helloWorldQueue")
private Queue helloWorldQueue;

@Inject
JMSContext context;

// Use the Queue object to send messages...
try {
    context.createProducer().send(helloWorldQueue, "Hello World!");
}
}
```



References

Further information is available in the Remote JNDI lookup chapter and the Contexts and Dependency Injection (CDI) chapter of the *Development Guide* for Red Hat JBoss EAP:

<https://access.redhat.com/documentation/en/red-hat-jboss-enterprise-application-platform/7.0/>

► Guided Exercise

Identifying JNDI Resources

In this exercise, you will explore the JNDI resource bindings in the application server.

Outcomes

You should be able to start EAP from a terminal window and explore the JNDI resource bindings in the application server.

Before You Begin

JBoss EAP is already installed in the `/opt/jboss-eap-7.0` folder and is accessible with the environment variable `JBOSS_HOME` that points to this folder.

Open a terminal window on the workstation VM and run the following command to verify that EAP is not currently running:

```
[student@workstation ~]$ lab jndi setup
```

Steps

- 1. Start JBoss EAP in a new terminal window.
 - 1.1. Open a terminal window on the workstation VM (Applications > Favorites > Terminal, or right-click the desktop and select **Open in Terminal**) and run the following commands to start EAP:

```
[student@workstation ~]$ cd $JBOSS_HOME/bin  
[student@workstation bin]$ ./standalone.sh -c standalone-full.xml
```



Note

Throughout this course, you will start EAP in stand-alone mode with the `standalone-full` profile, which contains all of the APIs and subsystems that support the Java EE 7 full profile.

- 1.2. As the application server is starting, the server prints log messages to the console where you started EAP. If EAP starts successfully, you see output similar to the following:

```
13:35:23,615 INFO [org.jboss.as] (Controller Boot Thread) WFLYSRV0051: Admin  
console listening on http://127.0.0.1:9990  
13:35:23,615 INFO [org.jboss.as] (Controller Boot Thread) WFLYSRV0025: JBoss EAP  
7.0.0.GA (WildFly Core 2.1.2.Final-redhat-1) started in 3321ms - Started 319 of  
604 services (393 services are lazy, passive or on-demand)
```

- 2. View the server log messages in the console as well as in the server log file.

- 2.1. EAP prints server log messages to the console where you started it. Briefly review the log messages by scrolling up and down in the terminal window.
- 2.2. Although viewing log messages in the console is useful for basic troubleshooting, EAP prints a lot of messages when multiple applications are deployed and viewing these messages in the console becomes unwieldy and difficult. To enable offline analysis, EAP also writes log messages to a file.

Open the `/opt/jboss-eap-7.0/standalone/log/server.log` file in a text editor. Briefly review the messages in the log file.

► 3. Explore JNDI resource bindings in the server log file.

- 3.1. The application server maintains a list of JNDI resource bindings. Resources that are needed by applications, for example, mail, JDBC data sources, and JMS connection factories and queues are bound to unique identifiable names under the respective name spaces.

JDBC data sources are bound to the `java:jboss/datasources/*` name space. In the `/opt/jboss-eap-7.0/standalone/log/server.log` file, verify that you can see the following two data source bindings:

```
WFLYJCA0001: Bound data source [java:jboss/datasources/ExampleDS]
WFLYJCA0001: Bound data source [java:jboss/datasources/MySQLDS]
```

**Note**

Hint: You can use your text editor's search capability (usually `Ctrl+F`) or use the `grep` command to find the bindings in the log file:

```
[student@workstation ~]$ grep -i
"datasources" /opt/jboss-eap-7.0/standalone/log/server.log
```

The `ExampleDS` binding points to an embedded H2 database that ships with EAP. The `MySQLDS` binding points to a MySQL database that will be used by the To Do List Java EE application that you will build during this course.

- 3.2. Explore the JMS-related JNDI bindings.

In the `/opt/jboss-eap-7.0/standalone/log/server.log` file, verify that you can see the following JMS-related bindings:

```
WFLYMSGAMQ0002: Bound messaging object to jndi name java:jboss/exported/jms/
RemoteConnectionFactory
...
WFLYMSGAMQ0002: Bound messaging object to jndi name java:/ConnectionFactory
...
WFLYMSGAMQ0002: Bound messaging object to jndi name java:jboss/
DefaultJMSConnectionFactory
```

► 4. Stop EAP by pressing `Ctrl+C` in the terminal window where you started it.

This concludes the guided exercise.

Packaging and Deploying a Java EE Application

Objective

After completing this section, students should be able to package a simple Java EE application and deploy it to JBoss EAP using Maven.

Packaging and Deploying Java EE Applications

Java EE applications can be packaged in different ways for deployment to a compliant application server. Depending on the application type and the components it contains, applications can be packaged into different deployment types (compressed archive files containing classes, application assets, and XML deployment descriptors). The three most common deployment types are:

- **JAR files:** JAR files can contain Plain Old Java Object (POJO) classes, JPA Entity Beans, utility Java classes, EJBs, and MDBs. When deployed into an application server, depending on the type of components inside the JAR file, the application server looks for XML deployment descriptors, or code-level annotations, and deploys each component accordingly.

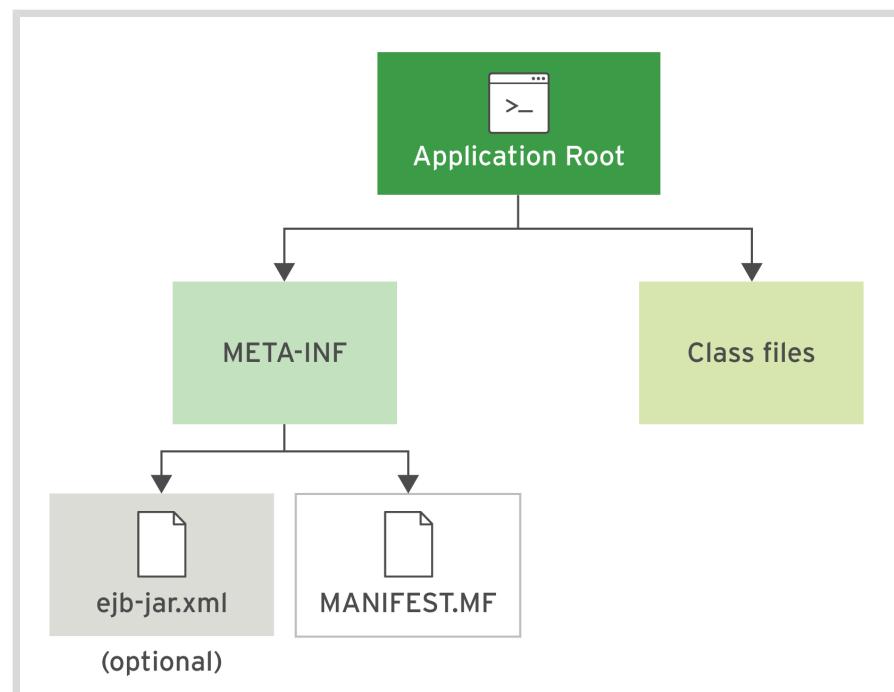


Figure 2.3: Sample EJB JAR file structure

- **WAR files:** A WAR file is used for packaging web applications. It can contain one or more JAR files, as well as XML deployment descriptor files under the WEB-INF or WEB-INF/classes/META-INF folders.

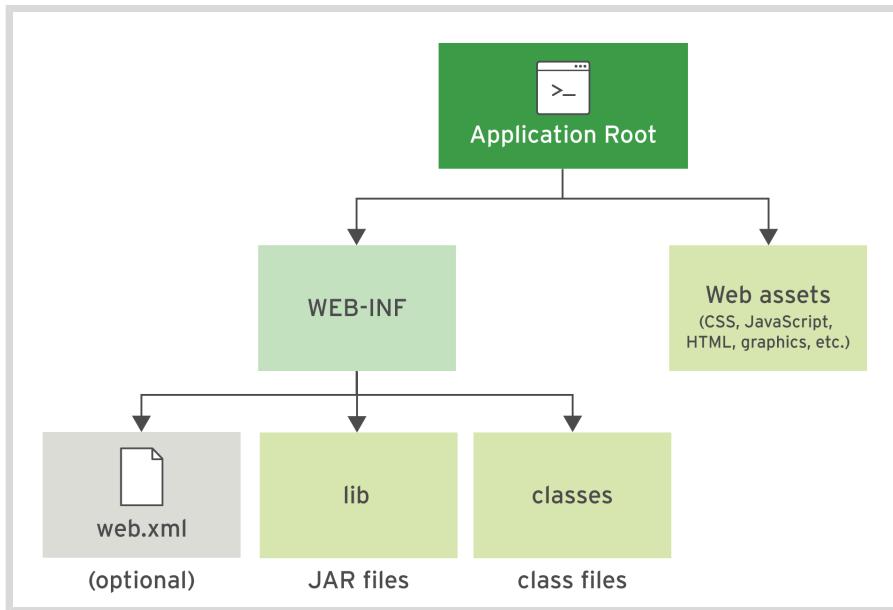


Figure 2.4: Sample WAR file structure

- **EAR files:** An EAR file contains multiple JAR and WAR files, as well as XML deployment descriptors in the META-INF folder.

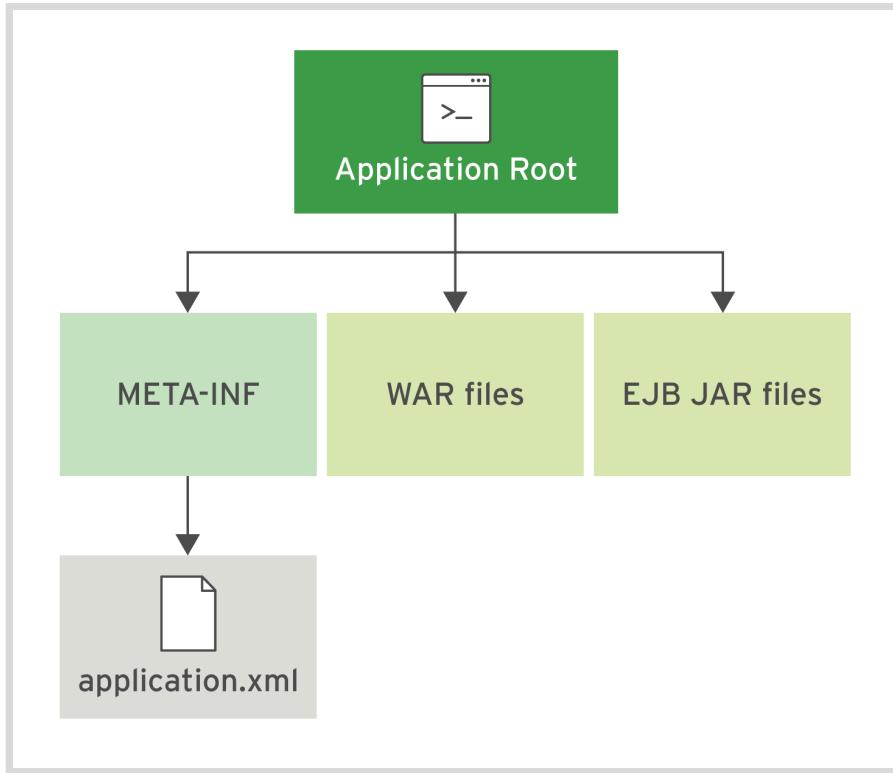


Figure 2.5: Sample EAR file structure

**Note**

XML deployment descriptors, if present, override the code-level annotations. For a given component, avoid duplicating the configuration in both places.

Packaging and Deploying Java EE Applications to EAP

Maven provides several useful plug-ins to simplify packaging and deployment to EAP during the development life cycle.

The `maven-war-plugin` creates WAR files from your application, provided you have followed the Maven standard source code layout. The `maven-war-plugin` can be declared in the `<build>` section of your Maven `pom.xml` file:

```
<build>
<finalName>todo</finalName>
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-war-plugin</artifactId>
    <version>${version.war.plugin}</version>
    <extensions>false</extensions>
    <configuration>
      <failOnMissingWebXml>false</failOnMissingWebXml>
    </configuration>
  </plugin>
</plugins>
</build>
```

Similarly, the `maven-ear-plugin` creates EAR files from your application source code. It is declared in the `<build>` section of your Maven `pom.xml` file. You need to indicate the WAR files that should be packaged inside the EAR file with the `<webModule>` tag:

```
<build>
<finalName>todo</finalName>
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-ear-plugin</artifactId>
    <version>${version.ear.plugin}</version>
    <configuration>
      <version>6</version>
      <defaultLibBundleDir>lib</defaultLibBundleDir>
      <modules>
        <webModule>
          <groupId>com.redhat.training</groupId>
          <artifactId>todojee-web</artifactId>
          <contextRoot>/todo-ear</contextRoot>
        </webModule>
      </modules>
      <fileNameMapping>no-version</fileNameMapping>
    </configuration>
  </plugin>
</plugins>
</build>
```

You can use Maven to deploy applications to JBoss EAP using the `wildfly-maven-plugin`, which provides features to deploy and undeploy applications to EAP. It supports deploying all

three types of deployment units: JAR, WAR, and EAR. You can declare the plug-in in your project's Maven pom.xml file:

```
<plugin>
  <groupId>org.wildfly.plugins</groupId>
  <artifactId>wildfly-maven-plugin</artifactId>
  <version>${version.wildfly.maven.plugin}</version>
</plugin>
```

To build, package, and deploy an application to EAP, run the following command from your project root folder:

```
[student@workstation todojee]$ mvn clean package wildfly:deploy
```

To undeploy an application from EAP, run the following command from your project root folder:

```
[student@workstation todojee]$ mvn wildfly:undeploy
```



References

Further information is available in the Deploying Applications Using Maven chapter

of the *Development Guide* for Red Hat JBoss EAP 7.0:

<https://access.redhat.com/documentation/en/red-hat-jboss-enterprise-application-platform/>

► Workshop

Packaging and Deploying a Java EE Application

Outcomes

You should be able to import a simple Java EE web application project into JBoss Developer Studio and package and deploy it to EAP.

Before You Begin

Open a terminal window on the workstation VM and run the following command to download the lab files that are required for this workshop.

```
[student@workstation ~]$ lab hello-web setup
```

- ▶ 1. Import the hello-web project into JBoss Developer Studio IDE (JBDS).
 - 1.1. Start JBDS by double-clicking the JBDS icon on the workstation VM desktop.
 - 1.2. In the **Eclipse Launcher** window, enter **/home/student/JB183/workspace** in the **Workspace** field, and then click **Launch**.

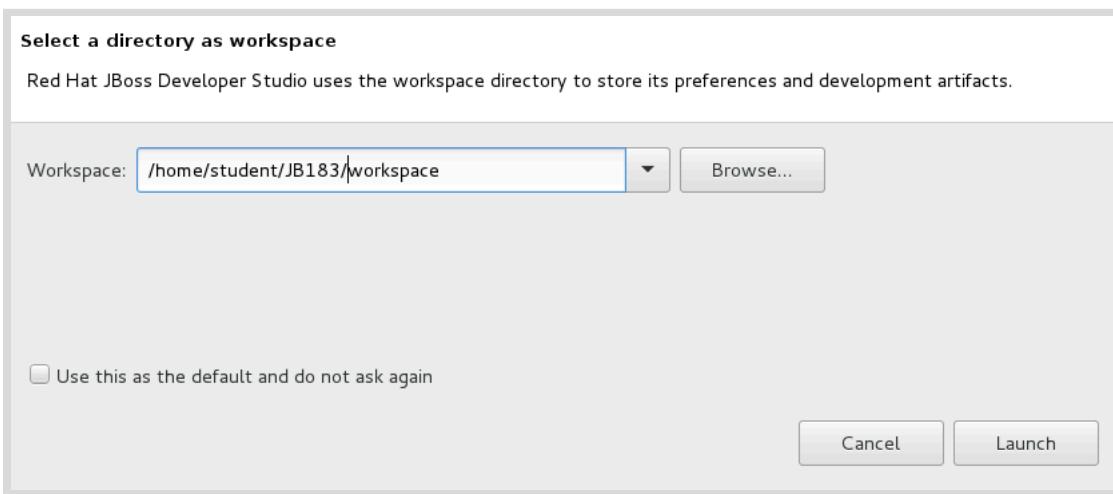


Figure 2.6: JBDS workspace dialog

- 1.3. In the JBDS menu, click **File > Import** to open the **Import** wizard.
- 1.4. On the **Select** page, click **Maven > Existing Maven Projects**, and then click **Next**.
- 1.5. In the **Maven projects** page, click **Browse** to open the **Select root folder** window. Navigate to the **/home/student/JB183/labs/** directory. Select the **hello-web** folder, and then click **OK**.
- 1.6. On the **Maven projects** page, click **Finish**.

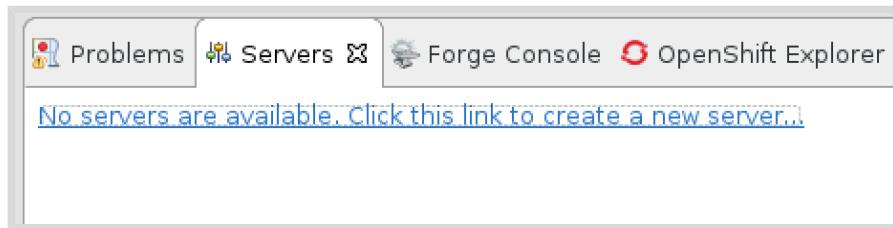
- 1.7. Watch the JBDS status bar to monitor the progress of the import operation. It may take a few minutes to download all the required dependencies.
- 2. Explore the project's Maven `pom.xml` file and the parent POM file.
- 2.1. Expand the **hello-web** item in the **Project Explorer** pane on the left and then double-click the `pom.xml` file.
- The **Overview** tab is visible in the main editor window, and displays a high-level view of the project. The **Group Id**, **Artifact Id**, and the **Version** (commonly referred to as the **GAV** coordinates of a project or module) of the **hello-web** project, as well as its parent, are shown in this tab.
- 2.2. Click the **Dependencies** tab to view the dependencies (libraries, frameworks, and modules that this project depends on) for the project.
 - 2.3. Click the **pom.xml** tab to view the full text of the `pom.xml` file.
- Briefly review the GAV coordinates for this project:
- ```
<artifactId>hello-web</artifactId>
<packaging>war</packaging>①
<name>Hello World web app Project</name>
<description>This is the hello-web project</description>
<parent>②
 <groupId>com.redhat.training</groupId>
 <artifactId>parent-pom</artifactId>
 <version>1.0</version>
 <relativePath>./pom.xml</relativePath>
</parent>
...
<build>
 <plugins>
 <plugin>③
 <artifactId>maven-war-plugin</artifactId>
 <version>${version.war.plugin}</version>
 ...
 </plugin>
 </plugins>
</build>
```
- ①** The packaging format is declared as `war`. Maven ensures that when the project is built, it will result in a WAR file that can be deployed to EAP.
  - ②** This project inherits the declarations and properties from the parent POM file, which is located at `/home/student/JB183/labs/pom.xml`. The parent POM file declares many attributes and properties that can be used by multiple child projects referencing it. It is a Maven best practice to declare repositories, master dependencies, bill of materials (BOM) declarations, and other attributes that are used in multiple projects, in order to avoid duplication.
  - ③** Because this project is a web application built as a WAR file, the Maven WAR plug-in (`maven-war-plugin`) is configured.
- 2.4. Open the `/home/student/JB183/labs/pom.xml` parent pom file using JBDS (click **File > Open File**), or in a text editor.
  - 2.5. The parent POM file declares a number of commonly used properties that are used in all the projects in this course. For example, the file declares that the version of the

JBoss EAP bill of materials (BOM) is 7.0.0.GA, and that all projects will be compiled with JDK 1.8.

The Wildfly Maven plug-in is also declared in the parent POM file. It is used to deploy the project WAR file to EAP and relies on the JBOSS\_HOME environment variable to identify the instance of JBoss EAP where the WAR file must be deployed.

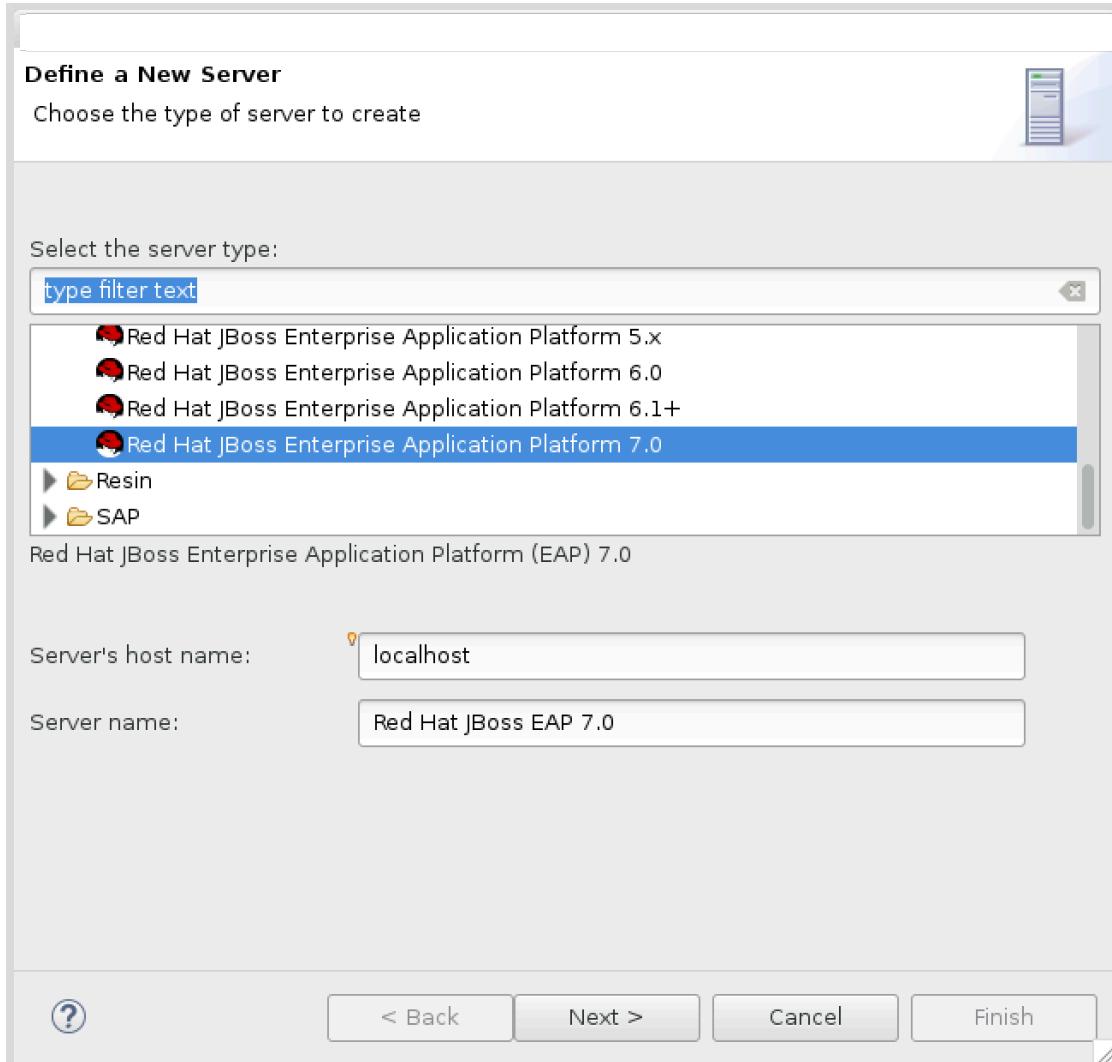
► **3.** Configure an EAP server instance in JBDS.

- 3.1. Click the **Servers** tab at the bottom of JBDS, below the main editor area.



**Figure 2.7: JBDS servers tab**

- 3.2. Click **No servers are available** to define a new EAP server.
- 3.3. In the **Define a New Server** window, select the **Red Hat JBoss Enterprise Application Platform 7.0** option and then click **Next**.



**Figure 2.8: Define a new EAP server**

- 3.4. In the **Create a new Server Adapter** window, leave the fields at their default values as shown in the figure below, and then click **Next**.

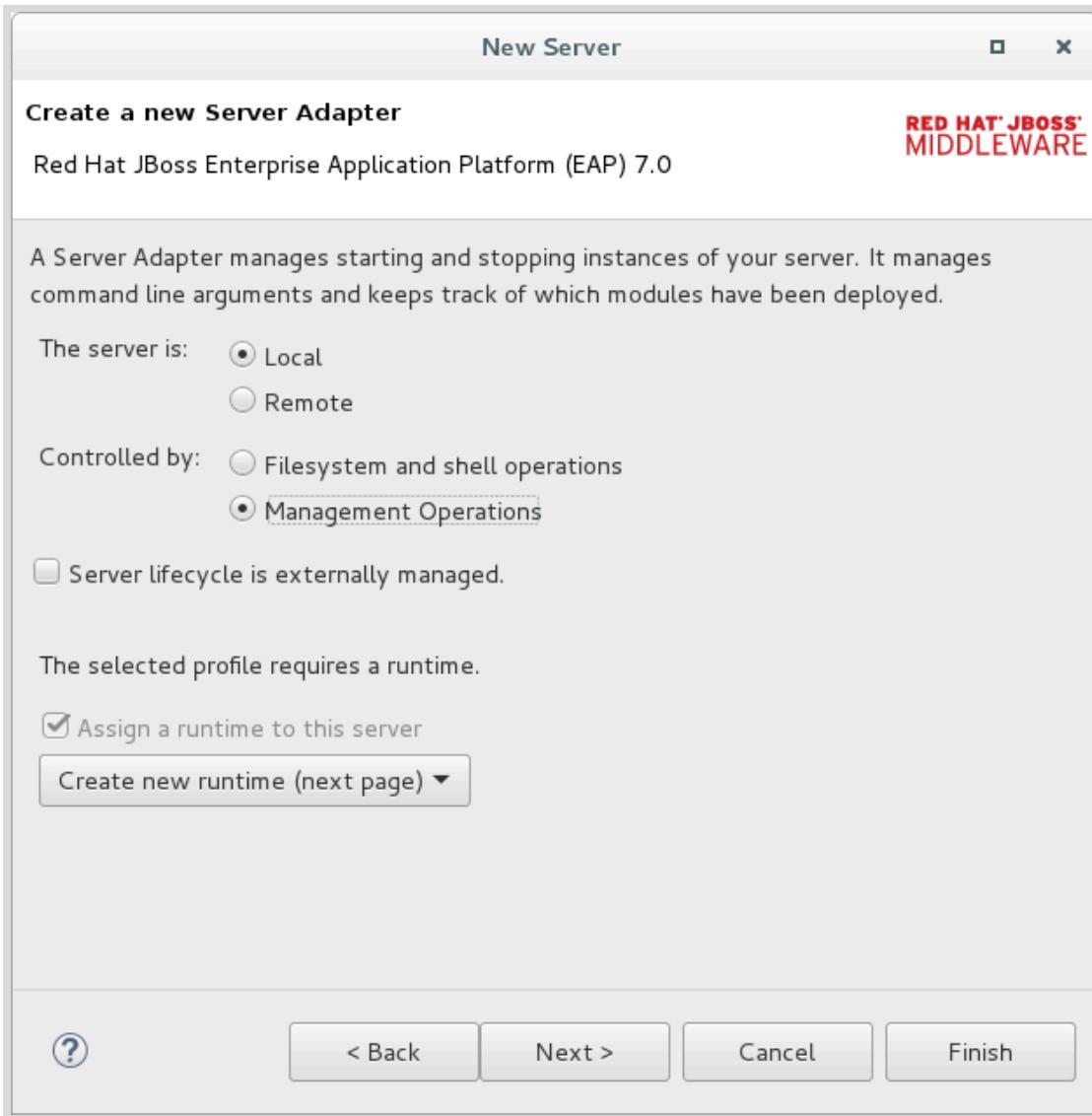


Figure 2.9: New server adapter configuration

- 3.5. In the **JBoss Runtime** window, click **Browse** next to the **Home Directory** field and select the `/opt/jboss-eap-7.0` folder (the environment variable `$JBOSS_HOME` points to this folder).  
Because you will be running the `standalone-full` profile, edit the **Configuration file** field and change it to `standalone-full.xml` from the default `standalone.xml`.  
Click **Next** to continue.

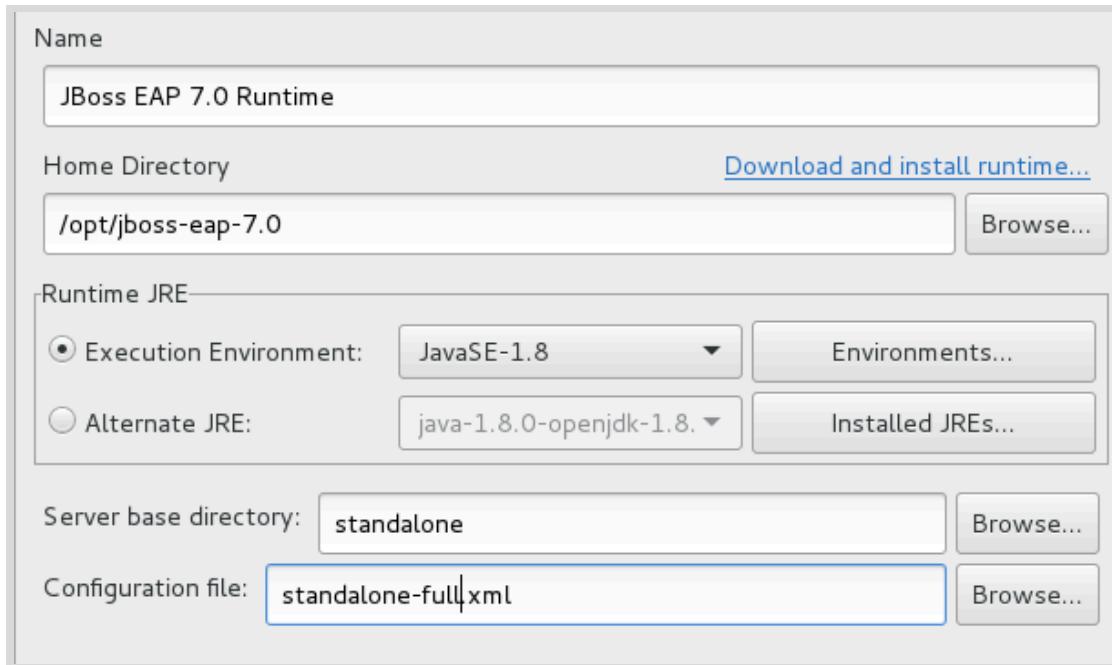


Figure 2.10: JBoss EAP runtime configuration

- 3.6. In the Add and Remove screen, click Finish.
- 3.7. You should now see a new server entry called **Red Hat JBoss EAP 7.0** added to the **Servers** tab of JBDS. Click this entry to expand it.

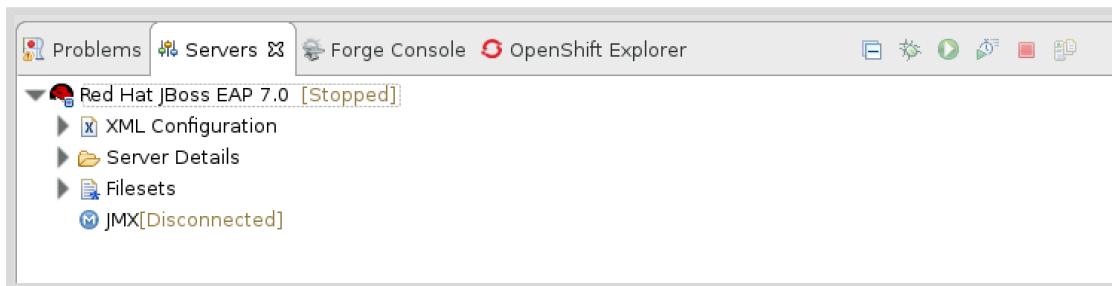


Figure 2.11: Adding a new JBoss EAP server

- 4. Start EAP from within JBDS.
- 4.1. Right-click **Red Hat JBoss EAP 7.0** in the **Servers** tab and then click **Start** (the green play icon) to start the newly added EAP instance.

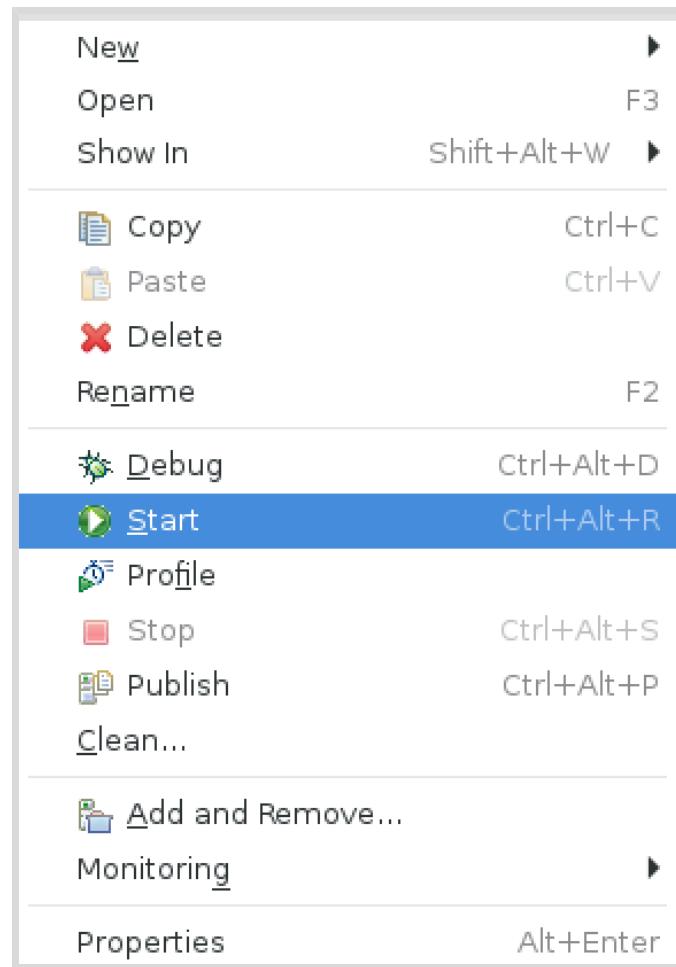


Figure 2.12: Starting a JBoss EAP server

- 4.2. As EAP starts, it prints messages to the **Console** tab of JBDS. Verify that no errors appear in the console.

The screenshot shows the JBDS (JBoss Developer Studio) interface with the 'Console' tab selected. The tab displays the following log output from a Red Hat JBoss EAP 7.0 server:

```

Red Hat JBoss EAP 7.0 [JBoss Application Server Startup Configuration] /usr/lib/jvm/java-1.8.0-openjdk-1.8.0
23:27:46,119 INFO [org.jboss.modules] (main) JBoss Modules version 1.5.1.Final-redhat-1
23:27:46,614 INFO [org.jboss.msc] (main) JBoss MSC version 1.2.6.Final-redhat-1
23:27:46,693 INFO [org.jboss.as] (MSC service thread 1-3) WFLYSRV0049: JBoss EAP 7.0
23:27:50,323 INFO [org.jboss.as.server] (Controller Boot Thread) WFLYSRV0039: Creating
23:27:50,369 INFO [org.xnio] (MSC service thread 1-1) XNIO version 3.3.6.Final-redhat-1
23:27:50,399 INFO [org.xnio.nio] (MSC service thread 1-1) XNIO NIO Implementation Version
23:27:50,568 INFO [org.jboss.remoting] (MSC service thread 1-1) JBoss Remoting version
23:27:50,588 INFO [org.jboss.as.clustering.infinispan] (ServerService Thread Pool -- 1)
23:27:50,811 WARN [org.jboss.as.txn] (ServerService Thread Pool -- 60) WFLYTX0013: M
23:27:50,834 INFO [org.jboss.as.webservices] (ServerService Thread Pool -- 62) WFLYW
23:27:50,844 INFO [org.jboss.as.security] (ServerService Thread Pool -- 59) WFLYSEC0001:
23:27:50,839 INFO [org.jboss.as.jsf] (ServerService Thread Pool -- 48) WFLYJSF0007:
23:27:50,852 INFO [org.jboss.as.connector.subsystems.datasources] (ServerService Thread
23:27:50,899 INFO [org.jboss.as.naming] (ServerService Thread Pool -- 52) WFLYNAM0001:
23:27:50,930 INFO [org.wildfly.extension.io] (ServerService Thread Pool -- 40) WFLYI
23:27:50,943 INFO [org.wildfly.iiop.openjdk] (ServerService Thread Pool -- 42) WFLYI
23:27:51,058 INFO [org.jboss.as.connector.subsystems.datasources] (ServerService Thread

```

Figure 2.13: JBoss EAP console log output

► 5. Build, package, and deploy the hello-web application.

- 5.1. Open a new terminal window on the **workstation** VM and run the following commands to compile, package, and deploy the hello-web application using Maven:

```
[student@workstation ~]$ cd /home/student/JB183/labs/hello-web
[student@workstation hello-web]$ mvn clean package wildfly:deploy
```

When you run the above command, you see output like the following:

```
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 24.160 s
[INFO] Finished at: 2016-11-20T01:08:05-05:00
[INFO] Final Memory: 34M/248M
[INFO] -----
```

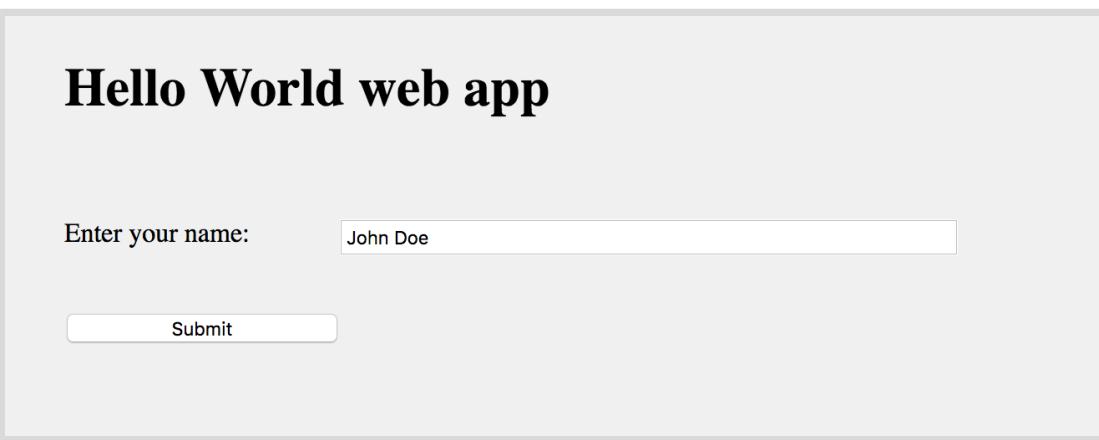
- 5.2. Click **Console** in JBDS for the EAP server and observe the hello-web application being deployed:

```
01:08:03,664 INFO [org.jboss.as.server.deployment] (MSC service thread 1-1)
WFLYSRV0027: Starting deployment of "hello-web.war" (runtime-name: "hello-
web.war")

...
01:08:05,624 INFO [org.wildfly.extension.undertow] (ServerService Thread Pool --
72) WFLYUT0021: Registered web context: /hello-web
01:08:05,705 INFO [org.jboss.as.server] (management-handler-thread - 1)
WFLYSRV0010: Deployed "hello-web.war" (runtime-name : "hello-web.war")
```

► 6. Access the hello-web application using a browser.

- 6.1. Verify that no errors appear in the console when the application is deployed. Use a web browser in the **workstation** VM to navigate to <http://localhost:8080/hello-web> to access the hello-web application.



**Figure 2.14: The hello-web application**

- ▶ 7. Enter **John Doe** in the **Enter your name** field and click **Submit**.
- ▶ 8. Verify that the server processes the input and responds with a Hello message, as well as the current time on the server.

The screenshot shows a web page with the title "Hello World web app". Below the title is a form with a label "Enter your name:" followed by an input field containing "John Doe". Below the input field is a "Submit" button. Underneath the form, there is a green bullet point followed by the text "Hello JOHN DOE!. Time on the server is: Nov 20 2016 01:09:55 AM".

Figure 2.15: The hello-web application response

- ▶ 9. Undeploy the application and stop EAP.
  - 9.1. In the terminal window where you ran the Maven command to deploy the application, run the following command to undeploy the application from EAP:

```
[student@workstation hello-web]$ mvn wildfly:undeploy
```

When you run the above command, the `hello-web.war` file is undeployed from EAP. You see the following output in the EAP Console:

```
21:00:31,705 INFO [org.wildfly.extension.undertow] (ServerService Thread Pool -- 77) WFLYUT0022: Unregistered web context: /hello-web
21:00:31,988 INFO [org.jboss.as.server] (management-handler-thread - 13) WFLYSRV0009: Undeployed "hello-web.war" (runtime-name: "hello-web.war")
```

- 9.2. Right-click on the `hello-web` project in the **Project Explorer**, and select **Close Project** to close this project.
- 9.3. Right-click **Red Hat JBoss EAP 7.0** in the **Servers** tab and then click **Stop** to stop the EAP instance.

This concludes the guided exercise.

## ► Lab

# Packaging and Deploying Applications to an Application Server

In this lab, you will learn how to package and deploy a Java EE application using Maven and JBoss Developer Studio.

## Outcomes

You should be able to package and deploy the To Do List Java EE application to JBoss EAP using Maven and JBoss Developer Studio.

## Before You Begin

Open a terminal window on the **workstation** VM and run the following command to download the files that are required for this lab.

```
[student@workstation ~]$ lab todojee setup
```

## Steps

1. Import the todojee project into JBoss Developer Studio IDE (JBDS).
2. Start EAP from within JBDS.
3. Build, package, and deploy the todo-app application to EAP.
4. Verify successful deployment of the todo-app WAR file on EAP.
5. Navigate to <http://localhost:8080/todo> to access the todo application.

## Task List

| ID | Description        | Done                                |                                    |
|----|--------------------|-------------------------------------|------------------------------------|
| 1  | Pick up newspaper  | <input type="checkbox"/>            | <span style="color: red;">X</span> |
| 2  | Buy groceries      | <input checked="" type="checkbox"/> | <span style="color: red;">X</span> |
| 3  | Pay telephone bill | <input type="checkbox"/>            | <span style="color: red;">X</span> |
| 4  | Buy milk           | <input type="checkbox"/>            | <span style="color: red;">X</span> |
| 5  | Buy butter         | <input checked="" type="checkbox"/> | <span style="color: red;">X</span> |

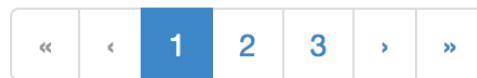


Figure 2.16: The To Do List Application

6. Explore the To Do List application functionality.

6.1. Browse existing tasks.

A set of tasks are preloaded into the database. Review the tasks by using the pagination bar below the list of tasks. The **Done** check box is selected for completed tasks, and the **Description** text appears in strikethrough typeface:

|   |               |                                     |                                    |
|---|---------------|-------------------------------------|------------------------------------|
| 2 | Buy groceries | <input checked="" type="checkbox"/> | <span style="color: red;">X</span> |
|   |               |                                     |                                    |

Figure 2.17: A completed task

6.2. Add a new task.

You can add a new task by entering a task description in the **Add Task** panel on the right side. If the task is already complete, you can select the **Completed** check box. Click **Save** to add your task to the database.

The screenshot shows a modal window titled "Add Task". Inside, there is a "Description:" label followed by a text input field containing "Read a Book", which has a small green checkmark icon to its right. Below this is a "Completed:" label with an empty square checkbox. At the bottom of the modal are two blue buttons: "Save" on the left and "Clear" on the right.

**Figure 2.18: Add a task**

Only five tasks are displayed per page. Use the pagination bar at the bottom of the table to navigate to the last page and verify that your newly added task is displayed.

- 6.3. Complete a task.

To complete a task, all you need to do is to toggle the checkbox in the **Done** column against each task. Toggling the checkbox again reverts the state and marks the task as pending.

- 6.4. Delete a task.

To delete a task, click the red X in the right-most column.

7. Cleanup and grade.

- 7.1. To verify that you have successfully deployed the application, open a new terminal window on your workstation VM and run the following command to grade this lab:

```
[student@workstation ~]$ lab todojee grade
```

If you see failures after running the command, look at the errors, troubleshoot the deployment, and fix the errors. Rerun the grading script and verify that it passes.

- 7.2. In the terminal window where you ran the Maven command to deploy the application, run the following command to undeploy the application from EAP:

```
[student@workstation todojee]$ mvn wildfly:undeploy
```

```
21:16:33,035 INFO [org.jboss.as.server] (management-handler-thread - 9)
WFLYSRV0009: Undeployed "todo.war" (runtime-name: "todo.war")
```

- 7.3. Right-click on the todojee project in the **Project Explorer**, and select **Close Project** to close this project.

- 7.4. Right-click **Red Hat JBoss EAP 7.0** in the **Servers** tab, and then click **Stop** to stop the EAP instance.

This concludes the lab.

## ► Solution

# Packaging and Deploying Applications to an Application Server

In this lab, you will learn how to package and deploy a Java EE application using Maven and JBoss Developer Studio.

## Outcomes

You should be able to package and deploy the To Do List Java EE application to JBoss EAP using Maven and JBoss Developer Studio.

## Before You Begin

Open a terminal window on the **workstation** VM and run the following command to download the files that are required for this lab.

```
[student@workstation ~]$ lab todojee setup
```

## Steps

1. Import the todojee project into JBoss Developer Studio IDE (JBDS).
  - 1.1. Click the JBDS icon on the **workstation** VM desktop to start JBDS.
  - 1.2. In the JBDS menu, click **File > Import** to open the **Import** wizard.
  - 1.3. On the **Select** page, select **Maven > Existing Maven Projects**, and then click **Next**.
  - 1.4. In the **Maven projects** page, click **Browse** to open the **Select root folder** window.
  - 1.5. Navigate to the **/home/student/JB183/labs/** directory. Select the **todojee** folder, and then click **OK**.
  - 1.6. On the **Maven projects** page, click **Finish**.
  - 1.7. Wait until JBDS finishes importing the project.
2. Start EAP from within JBDS.
  - 2.1. Right-click **Red Hat JBoss EAP 7.0** in the **Servers** tab and then click **Start** to start the EAP instance that is configured in the previous exercise.
  - 2.2. As EAP starts, it prints messages to the **Console** tab of JBDS. Verify that there are no startup errors seen in the console.
3. Build, package, and deploy the todo-app application to EAP.

Open a new terminal window on your workstation and run the following commands:

```
[student@workstation ~]$ cd /home/student/JB183/labs/todojee
[student@workstation todojee]$ mvn clean package wildfly:deploy
```

```
...
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 9.295 s
[INFO] Finished at: 2016-11-21T16:05:11+05:30
[INFO] Final Memory: 27M/389M
[INFO] -----
```

- Verify successful deployment of the todo-app WAR file on EAP.

Change to the **Console** tab of JBDS and observe the todo-app application being deployed. Verify that no errors appear in the console:

```
WFLYSRV0027: Starting deployment of "todo.war" (runtime-name: "todo.war")
WFLYUT0021: Registered web context: /todo
WFLYSRV0010: Deployed "todo.war" (runtime-name : "todo.war")
```

- Navigate to <http://localhost:8080/todo> to access the todo application.

The screenshot shows a web application titled "Task List". Below the title is a table with five rows of tasks. The columns are labeled "ID", "Description", and "Done". The tasks are:

| ID | Description        | Done                                |
|----|--------------------|-------------------------------------|
| 1  | Pick up newspaper  | <input type="checkbox"/>            |
| 2  | Buy groceries      | <input checked="" type="checkbox"/> |
| 3  | Pay telephone bill | <input type="checkbox"/>            |
| 4  | Buy milk           | <input type="checkbox"/>            |
| 5  | Buy butter         | <input checked="" type="checkbox"/> |

Below the table is a navigation bar with buttons for page numbers: «, <, 1, 2, 3, >, ». The number 1 is highlighted in blue, indicating it is the current page.

**Figure 2.17: The To Do List Application**

- Explore the To Do List application functionality.

- Browse existing tasks.

A set of tasks are preloaded into the database. Review the tasks by using the pagination bar below the list of tasks. The **Done** check box is selected for completed tasks, and the **Description** text appears in strikethrough typeface:

|   |               |                                     |                                  |
|---|---------------|-------------------------------------|----------------------------------|
| 2 | Buy groceries | <input checked="" type="checkbox"/> | <input type="button" value="X"/> |
|---|---------------|-------------------------------------|----------------------------------|

Figure 2.18: A completed task

## 6.2. Add a new task.

You can add a new task by entering a task description in the **Add Task** panel on the right side. If the task is already complete, you can select the **Completed** check box. Click **Save** to add your task to the database.

**Add Task**

**Description:** Read a Book 

**Completed:**

**Save** **Clear**

Figure 2.19: Add a task

Only five tasks are displayed per page. Use the pagination bar at the bottom of the table to navigate to the last page and verify that your newly added task is displayed.

## 6.3. Complete a task.

To complete a task, all you need to do is to toggle the checkbox in the **Done** column against each task. Toggling the checkbox again reverts the state and marks the task as pending.

## 6.4. Delete a task.

To delete a task, click the red X in the right-most column.

## 7. Cleanup and grade.

## 7.1. To verify that you have successfully deployed the application, open a new terminal window on your workstation VM and run the following command to grade this lab:

```
[student@workstation ~]$ lab todojee grade
```

If you see failures after running the command, look at the errors, troubleshoot the deployment, and fix the errors. Rerun the grading script and verify that it passes.

## 7.2. In the terminal window where you ran the Maven command to deploy the application, run the following command to undeploy the application from EAP:

```
[student@workstation todojee]$ mvn wildfly:undeploy
```

```
21:16:33,035 INFO [org.jboss.as.server] (management-handler-thread - 9)
WFLYSRV0009: Undeployed "todo.war" (runtime-name: "todo.war")
```

- 7.3. Right-click on the **todojee** project in the **Project Explorer**, and select **Close Project** to close this project.
- 7.4. Right-click **Red Hat JBoss EAP 7.0** in the **Servers** tab, and then click **Stop** to stop the EAP instance.

This concludes the lab.

# Summary

---

In this chapter, you learned:

- An *application server* provides the necessary runtime environment and infrastructure to host and manage Java EE enterprise applications.
- *Red Hat JBoss EAP 7* is a Java EE 7 compliant application server that provides a reliable, high-performance, light-weight, and supported infrastructure for deploying Java EE applications.
- A *profile* in the context of a Java EE application server is a set of related APIs that target a specific application type. The Java EE 7 specification defines two profiles: *web* and *full*. JBoss EAP fully supports both profiles.
- A *resource* is a logical object that can be looked up and used by components in a Java EE application. Each resource is identified by a unique name, called the *JNDI name* or a *JNDI resource binding*.
- The JNDI bindings are organized under a logical *namespace*, in a tree-like hierarchy, usually called the *JNDI tree*.
- You can inject JNDI resources into applications that require the resource by using a `@Resource` annotation.
- Java EE applications are packaged and deployed in different formats. The three most common are: JAR, WAR, and EAR files.
- Maven deploys applications to JBoss EAP using the Wildfly Maven plug-in, which provides features to deploy and undeploy applications to EAP. It supports deploying all three types of deployment units: JAR, WAR, and EAR.



## Chapter 3

# Creating Enterprise Java Beans

### Overview

**Goal** Create Enterprise Java Beans.

**Objectives**

- Convert a POJO to an EJB.
- Access an EJB both locally and remotely.
- Describe the life cycle of EJBs.
- Describe container-managed and bean-managed transactions and demarcate each in an EJB.

**Sections**

- Converting a POJO to an EJB (and Guided Exercise)
- Accessing EJBs Locally and Remotely (and Guided Exercise)
- Describing the Life Cycle of an EJB (and Quiz)
- Demarcating Implicit and Explicit Transactions (and Quiz)

**Lab**

- Creating Enterprise Java Beans

# Converting a POJO to an EJB

## Objective

After completing this section, students should be able to convert a POJO to an EJB.

## Describing Enterprise Java Beans (EJB)

An *Enterprise Java Bean* (EJB) is a Java EE component typically used to encapsulate business logic in an enterprise application. Unlike simple Java beans in Java SE, where concepts such as multi-threading, concurrency, transactions, and security have to be explicitly implemented by the developer, in an EJB, the application server provides these features at runtime and enables the developer to focus on writing the business logic for the application.

Using EJBs to model the business logic of an enterprise application has several advantages:

- EJBs provide low-level system services, such as multi-threading and concurrency, without requiring the developer to write code explicitly for these services. This is important for enterprise applications with a large number of users accessing the application concurrently.
- The business logic is encapsulated into a portable component that can be distributed across many machines in a way that is transparent to clients and enables you to load balance requests when a large number of clients access the application concurrently.
- Client code is simplified because the client can focus on just the user-interface aspects without mixing business logic. For example, consider how the To Do List Java SE application combines both user-interface code and the core logic for list management in the same process and often in the same class.
- EJBs provide transactional capabilities to enterprise applications, where a number of users concurrently access the application and the application server ensures data integrity with the use of transactions.
- EJB components can be secured for access on a group or role basis. The application server provides an API for authentication and authorization services, without requiring the developer to write code explicitly.
- EJBs can be accessed by multiple different types of clients, ranging from stand-alone remote clients, other Java EE components, or web service clients using standard protocols like SOAP or REST.

## Reviewing the Types of EJB

The Java EE specification defines two different types of EJBs:

- **Session:** Performs an operation when called from a client. Usually an application's core business logic is exposed as a high-level API (*Session Facade* pattern) that can be distributed and can be accessed over a number of protocols (RMI, JNDI, web services).
- **Message Driven Bean (MDB):** Used for asynchronous communication between components in a Java EE application and can be used to receive Java Messaging Service (JMS) compatible messages and take some action based on the content of the received messages.

## Describing Session Beans

A Session Bean provides an interface to clients and encapsulates business logic methods that can be invoked by multiple clients either locally or remotely over different protocols. Session EJBs can be clustered and deployed across multiple machines in a client transparent manner. The Java EE standard does not formally define the low-level details of how EJBs should be clustered. Each application server provides its own mechanisms for clustering and high availability. A session bean's interface usually exposes a high-level API encapsulating the core business logic of the application.

There are **three** different types of session beans, depending on the application use case, that can be deployed on a Java EE compatible application server:

### Stateless Session Beans (SLSB)

A stateless session bean does not maintain conversational state with clients between calls. When clients interact with the stateless session bean and invoke methods on it, the application server allocates an instance from a pool of stateless session beans, which are pre-instantiated. Once a client completes the invocation and disconnects, the bean instance is either released back into the pool or destroyed.

A stateless session bean is useful in scenarios where the application has to serve a large number of clients concurrently accessing the bean's business methods. They typically can scale better than stateful session beans since the application server does not have to maintain state and the beans can be distributed across multiple machines in a large deployment.

Note that when working with stateless EJBs, you must be careful not to define stateful data elements and constructs that need to be shared between multiple clients (for example, map-like data structures holding a cache). These types of use cases would be more appropriately solved by using a stateful session bean or a singleton session bean.

A stateless session bean is also the preferred option for exposing SOAP or REST service endpoints to web services clients. Simple annotations are added to the bean class and methods to achieve this functionality without writing boilerplate code for web service communication.

### Stateful Session Beans (SFSB)

In contrast to stateless session beans, stateful session beans maintain conversational state with clients across multiple calls. There is a one-to-one relationship between the number of stateful bean instances and the number of clients. When a client completes the interaction with the bean and disconnects, the bean instance is destroyed. A new client results in a new stateful bean with its own unique state. The application server ensures that each client receives the same instance of a stateful session bean for each method call.

Stateful session beans are used in scenarios where conversational state has to be maintained with a client for the duration of the interaction. For example, a shopping cart bean tracks the number of items that a customer adds to their cart in an e-commerce application. Each customer's cart is encapsulated in the state of the bean and the state is updated as and when the customer adds, updates, or removes shopping items.

When a developer builds a stateful EJB, any class level attributes must be scoped as `private` and getter and setter methods are created to provide access to these attributes. This is a common pattern used in Java development but is also automatically incorporated when working with EJBs backing JSF (Java Server Faces) pages. When using expression language (EL) in the JSF source code to map form fields to EJB attributes, the EJB attributes are automatically accessed via getters and setters without explicitly using the method name.

An example of a stateful session bean is shown below with its getter and setter methods:

```

@Stateful
@Named("hello")
public class Hello {

 private String name;

 @Inject
 private PersonService personService;

 public void sayHello() throws IllegalStateException, SecurityException,
SystemException {
 String response = personService.hello(name);
 FacesContext.getCurrentInstance().addMessage(null, new FacesMessage(response));
 }

 public String getName() {
 return name;
 }

 public void setName(String name) {
 this.name = name;
 }

}

```

## Singleton Session Beans

Singleton session beans are session beans that are instantiated once per application and exists for the lifecycle of the application. Every client request for a singleton bean goes to the same instance. Singleton session beans are used in scenarios where a single enterprise bean instance is shared across multiple clients.

Unlike stateless session beans, which are pooled by the application server, there is only a single instance of a singleton session bean in memory. Similar to stateless session beans, singleton session beans can also be used for implementing web service endpoints. A developer can provide annotations to indicate that the bean must be initialized by the application server at startup as a performance optimization (for example, database connections, JNDI lookups, JMS remote connection factory creation, and many more).

## Message Driven Beans

A Message Driven Bean (MDB) enables Java EE applications to process *messages* asynchronously. Once deployed on an application server, it listens for JMS messages and for each message received, it performs an action (the `onMessage()` method of the MDB is invoked). MDBs provide an event driven *loosely coupled* model for application development. The MDBs are not injected into or invoked from client code but are triggered by the receipt of messages.

The MDB is stateless and does not maintain any conversational state with clients. The application server maintains a pool of MDBs and manages their lifecycle by assigning and returning instances from and to the pool. They can also participate in transactions and the application server takes care of message redelivery and message receipt acknowledgment based on the outcome of the message processing.

There are numerous use cases where MDBs can be used. The most popular is for decoupling systems and preventing their APIs from being too tightly coupled by direct invocation. Instead, two systems can communicate by passing messages in an asynchronous manner, which ensures that the two systems can independently evolve without impacting each other.

## Generating an EJB Automatically Using JBoss Developer Studio

There are a number of templates provided by JBDS that can be leveraged to automatically generate code. Using a template, it is possible to leverage JBDS to generate the shell of an EJB automatically. To accomplish this, the following steps must be followed:

1. In Project Explorer pane on the left side of JBDS, select the project you want to add an EJB class to, then right-click on the project name. Select **New** and then scroll to the bottom and select **Other**.
2. Once the search pane opens, navigate to **EJB** and choose **Session Bean (EJB 3.x)**.

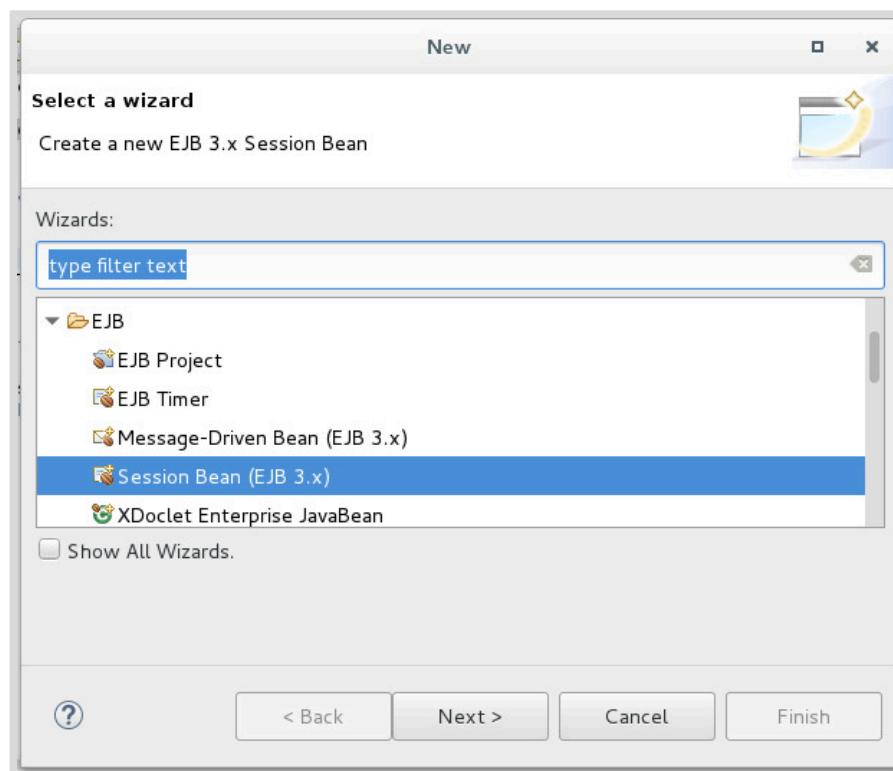


Figure 3.1: Create a new EJB in JBDS

3. Provide the package name, as well as the class name for the EJB class. Also, specify the state, then click **Next**.

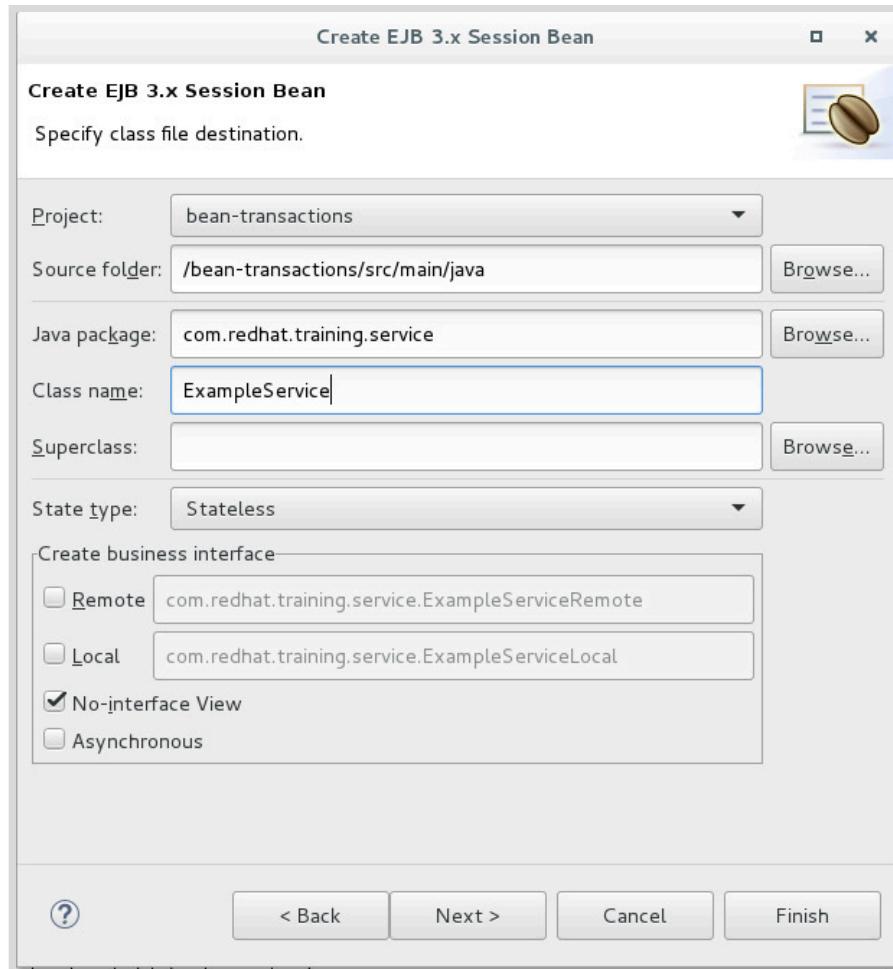


Figure 3.2: Name the new EJB Java class in JBDS and set its state

4. Optionally, specify an alternative name to be used when injecting this EJB, as well as the transaction type for the EJB, and then click Finish.

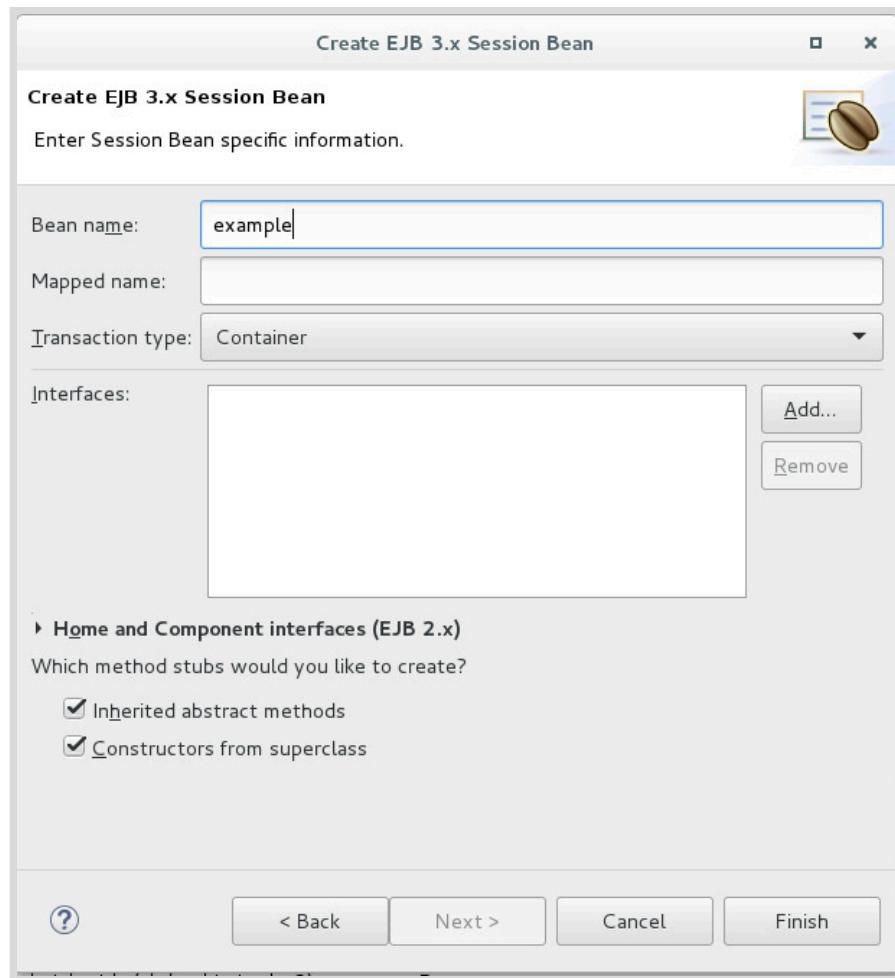


Figure 3.3: Provide a name for the EJB and setting its transaction type

5. The new EJB class opens in the editor window.

```

1 package com.redhat.training.service;
2
3 import javax.ejb.LocalBean;
4
5 /**
6 * Session Bean implementation class ExampleService
7 */
8
9 @Stateless(name = "example")
10 @LocalBean
11 public class ExampleService {
12
13 /**
14 * Default constructor.
15 */
16 public ExampleService() {
17 // TODO Auto-generated constructor stub
18 }
19
20 }
21

```

Figure 3.4: The new EJB class, automatically created.

## Converting a POJO to an EJB

Converting a POJO to an EJB is a simple process of annotating the POJO with one or more annotations defined in the Java EE standard and running the resultant EJB in the context of an application server. Take the case of a To Do List application POJO:

```
public class TodoBean {

 public void addTodo(TodoItem item) {
 ...
 }

 public void findTodo(int id) {
 ...
 }

 public void updateTodo(TodoItem item) {
 ...
 }

 public void deleteTodo(int id) {
 ...
 }
}
```

The POJO has four business methods to add, find, update, and delete to do items. To convert this POJO to a stateless session EJB is as simple as adding an `@Stateless` annotation to the POJO.

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

To convert this POJO to a stateful session bean, add the `@Stateful` annotation:

```
@Stateful
public class TodoBean {
 ...
}
```

In both the above cases, the application server automatically ensures that the methods in the EJB execute in a transactional context. You can further annotate the EJB with security-related annotations and expose the EJB as a web service end-point by adding web services annotations from the Java EE standard.

To convert this POJO to a singleton session bean, add the `@Singleton` annotation:

```
@Singleton
public class TodoBean {
 ...
}
```

In scenarios where you want a singleton bean to perform some initialization before starting to service client requests, you can add the `@Startup` annotation to the singleton class to tell the EJB container this class is required during the application initialization sequence and should be created first, before any other EJBs are instantiated. It is important to note that the application will fail to start if any EJB marked with `@Startup` throws an exception during initialization.

It is also possible to annotate an initialization method with the `@PostConstruct` annotation, which tells the EJB container to call that method immediately after instantiating the EJB.

The following example shows an EJB that is initialized for application startup and uses the `init()` method to setup its initial state:

```
@Singleton
@Startup
public class TodoBean {

 @PostConstruct
 public void init() {
 // do some initialization
 }
 ...
}
```

Another important distinction between POJOs and EJBs is that, because EJBs are instantiated by the EJB container, they cannot use a constructor that relies on arguments. This is because the EJB container cannot appropriately set these arguments when instantiating an instance of the EJB.

For this reason, if you are working on converting a POJO class that currently uses a constructor with arguments into an EJB, you need to find a way to provide equivalent logic in an argument-free constructor. If no constructor is provided for an EJB class, the EJB container uses the default no-argument constructor provided by the JVM. If an EJB class provides only a constructor with arguments, an error is raised by the container during application deployment.

## Demonstration: Converting a POJO to an EJB

1. Run the following command to prepare files used by this demonstration.

```
[student@workstation ~]$ demo convert-ejb setup
```

2. Start JBDS and import the `convert-ejb` project.

This project is a simple web-app using a JSF page backed by a stateful request-scoped EJB to provide a random answer to a question.

3. Inspect the `pom.xml` file, and observe the dependency on the EJB specification.

```
<dependency>
 <groupId>javax.enterprise</groupId>
 <artifactId>cdi-api</artifactId>
 <scope>provided</scope>
</dependency>

<dependency>
 <groupId>org.jboss.spec.javaee</groupId>
 <artifactId>jboss-ejb-api_3.2_spec</artifactId>
 <scope>provided</scope>
</dependency>
```

Both of these libraries are required to use the CDI and EJB annotations, but can be marked as provided because they are located on the application server and are available at runtime without being packaged in the WAR file.

4. Review the JSF page `src/main/web-app/index.xhtml`, which defines the view the user sees in their browser.

```
<ui:define name="content">
 <h1>Magic 8 Ball web app</h1>
 <br class="clear"/>
 <h:form id="form">
 <p class="input">
 <h:outputLabel value="Enter your question:" for="question" />
 <h:inputText value="#{eightBall.question}" id="question" required="true"
 requiredMessage="Question is required"/>
 </p>
 <br class="clear"/>
 <br class="clear"/>
 <p class="input">
 <h:commandButton action="#{eightBall.answerQuestion()}" value="Submit"
 styleClass="btn" />
 </p>
 <br class="clear"/>
 <br class="clear"/>
 <h:messages styleClass="messages"/>
 </h:form>
</ui:define>
```

JSF uses expression language (EL) to connect elements in the UI to methods on an EJB. For example, the EL expression `#{eightBall.question}` automatically uses the getter and setter for the `question` field on the EJB class `EightBall.java`.

Another example of EL is the expression `#{eightBall.answerQuestion()}`, which maps to the `answerQuestion()` method on the `EightBall` EJB class.

5. Update the `EightBall` managed bean class used by the JSF page.

```
//TODO Make @RequestScoped
//TODO Name "eightBall"
public class EightBall {

 private String question;

 //TODO Inject this EJB

 Magic8BallBean eightBallEJB;

 public String ask() {
 return eightBallEJB.answerQuestion(question);
 }
 public void answerQuestion() {
 String response = ask();
 FacesContext.getCurrentInstance().addMessage(null, new FacesMessage(response));
 }
}
```

```

public String getQuestion() {
 return question;
}

public void setQuestion(String question) {
 this.question = question;
}

}

```

This bean is `@RequestScoped` because it holds the current value of the `question` variable, which changes with each request or form submission.

Update the class level annotations to include `@RequestScoped` and `@Named("eightBall")` annotations to turn the `EightBall` class into an EJB that is available to be used by the JSF page.

```

@RequestScoped
@Named("eightBall")
public class EightBall {

```

Include an `@EJB` annotation so that the `Magic8BallBean` is injected into this bean.

```

@EJB
Magic8BallBean eightBallEJB;

```

6. Update the `Magic8BallBean` class to be a stateless EJB so that it can be used by the `EightBall` EJB.

```

//TODO Make this a stateless EJB
public class Magic8BallBean {

 public String answerQuestion(String question) {

 String[] answers = new String[10];
 answers[0] = "It is decidedly so.";
 answers[1] = "Ask again later.";
 answers[2] = "My reply is no.";
 answers[3] = "Cannot predict now.";
 answers[4] = "Don't count on it.";
 answers[5] = "As I see it, yes.";
 answers[6] = "Signs point to yes.";
 answers[7] = "My sources say no.";
 answers[8] = "Yes.";
 answers[9] = "No.";

 String answer = answers[ThreadLocalRandom.current().nextInt(0, 10)];

 // respond back with Question: {question} | Answer: {answer}.
 return "Question: " + question + " | Answer: " + answer;
 }
}

```

Update the class level annotation to include `@Stateless`. This makes the EJB available for injection.

```
//TODO Make this a stateless EJB
@Stateless
public class Magic8BallBean {
```

- Start the local JBoss EAP server inside JBDS.

In the **Servers** tab in the bottom pane of JBDS, right-click the server **Red Hat JBoss EAP 7.0 [Stopped]** and click on the green start button to start the server. Watch the **Console** until the server starts and you see the started message:

```
INFO [org.jboss.as] (Controller Boot Thread) WFLYSRV0025: JBoss EAP 7.0.0.GA
(WildFly Core 2.1.2.Final-redhat-1) started
```

The server must be started in order to run any Arquillian tests because they require the EJB container provided by JBoss EAP.

- Run the provided JUnit test `test/java/com/redhat/training/ejb/EJBTest.java` to verify that the bean injection is working properly by right-clicking on the `EJBTest.java` file and clicking **Run As** and then clicking **JUnit Test**.

```
@RunWith(Arquillian.class)
public class EJBTest {
 @Inject
 private EightBall eightBall;

 @Deployment
 public static WebArchive createDeployment() {
 return ShrinkWrap.create(WebArchive.class, "convert-ejb-
test.war").addClass(EightBall.class).addClass(Magic8BallBean.class)
 .addAsManifestResource(EmptyAsset.INSTANCE,
 ArchivePaths.create("beans.xml"));
 }

 @Test
 public void testEightBallEJB() {
 eightBall.setQuestion("Is it going to rain today?");
 Assert.assertNotNull(eightBall.ask());
 }
}
```

The test class uses Arquillian to run on the server. This is necessary because a container is required for CDI to function properly. Make sure the test passes to verify that the backing bean and the EJB are correctly injected.

- Deploy the application to the local JBoss EAP server, and test it in a browser. Run the following command in a terminal window:

```
[student@workstation ~]$ cd /home/student/JB183/labs/convert-ejb
[student@workstation convert-ejb]$ mvn wildfly:deploy
```

Open the following URL in your browser `http://localhost:8080/convert-ejb`, then test the application and make sure it is properly answering questions.

10. Undeploy the application and stop the server.

```
[student@workstation convert-ejb]$ mvn wildfly:undeploy
```



### References

Further information is available in the Session Beans chapter and the Message Driven Beans (MDB) chapter of the *Development Guide* for Red Hat JBoss EAP:  
<https://access.redhat.com/documentation/en/red-hat-jboss-enterprise-application-platform/7.0/>

## ► Guided Exercise

# Creating a Stateless EJB

In this exercise, you will create a stateless EJB that will be invoked and the results will be displayed on a web page.

## Outcomes

You should be able to implement a stateless EJB that can be invoked from a JSF managed bean.

## Before You Begin

Open a terminal window on the **workstation** VM and run the following command to download the files required for this lab.

```
[student@workstation ~]$ lab stateless-ejb setup
```

## Steps

- ▶ 1. Open JBDS and import the Maven project.
  - 1.1. Open JBDS by double-clicking the JBoss Developer Studio icon from the workstation desktop. Set the workspace to `/home/student/JB183/workspace` and click **OK**.
  - 1.2. In the JBDS menu, click **File > Import** to open the **Import** wizard.
  - 1.3. On the **Select** page, click **Maven > Existing Maven Projects**, and then click **Next**.
  - 1.4. In the **Maven projects** page, click **Browse** to open the **Select root folder** window. Navigate to the `/home/student/JB183/labs/` directory. Select the `stateless-ejb` folder, and then click **OK**.
  - 1.5. On the **Maven projects** page, click **Finish**.
  - 1.6. Watch the JBDS status bar to monitor the progress of the import operation. It may take a few minutes to download all the required dependencies.
- ▶ 2. Explore the project's `pom.xml` file by expanding the `stateless-ejb` item in the **Project Explorer** tab in the left pane of JBDS, and double-clicking the `pom.xml` file.
  - 2.1. Click on the **Overview** tab in the main editor window. This tab shows a high-level view of the project, and any changes made to this window will get applied to the appropriate section of the `pom.xml` file.
  - 2.2. Click the **Dependencies** tab to view the dependencies (libraries, frameworks and modules that this project depends on) for the project.
  - 2.3. Click the `pom.xml` tab to view the full text of the `pom.xml` file.

Observe that the EJB API is declared as a dependency with scope as provided. This is because JBoss EAP implements the complete Java EE profile, and therefore provides the necessary EJB libraries at runtime.

```
<dependency>
 <groupId>org.jboss.spec.javaee</groupId>
 <artifactId>jboss-ejb-api_3.2_spec</artifactId>
 <scope>provided</scope>
</dependency>
```

This also tells Maven not to package these libraries into the final WAR file.

### ► 3. Explore the application source code.

- 3.1. Review the JSF page that calls the EJB by expanding the **stateless-ejb** item in the **Project Explorer** tab in the left pane of JBDS. Further expand the **stateless-ejb > src > main > webapp** folders and double-click the **index.xhtml** file.

```
<h:form id="form">
 <p class="input">
 <h:outputLabel value="Enter your name:" for="name" />
 <h:inputText value="#{hello.name}" id="name" required="true"
 requiredMessage="Name is required"/>
 </p>
 <br class="clear"/>
 <br class="clear"/>
 <p class="input">
 <h:commandButton action="#{hello.sayHello()}" value="Submit" styleClass="btn" />
 </p>
 <br class="clear"/>
 <br class="clear"/>
 <h:messages styleClass="messages"/>
</h:form>
```

The Expression Language (EL) value `#{hello.sayHello()}`, is invoked upon web form submission.



#### Note

To view the actual source of the `index.xhtml` file, click the **Source** tab.

- 3.2. Explore the `Hello.java` Java file.

In the expanded **stateless-ejb** item in the **Project Explorer** tab in the left pane of JBDS, select **stateless-ejb > Java Resources > src/main/java > com.redhat.training.ui** and expand it. Double-click the `Hello.java` file.

Observe that the stateless EJB is injected using the `@EJB` annotation.

```
@EJB
private HelloBean helloEJB;
```

- 3.3. Explore the `HelloBean.java` Java file.

In the expanded **stateless-ejb** item in the **Project Explorer** tab in the left pane of JBDS, select **stateless-ejb > Java Resources > src/main/java > com.redhat.training.ejb** and expand it. Double-click the **HelloBean.java** file.

```
public class HelloBean {

 public String sayHello(String name) {

 // respond back with Hello, {name}.
 return "Hello, " + name;
 }
}
```

This bean defines the public method **sayHello**, which simply echoes back a string that is sent as input.

▶ **4.** Start EAP.

Select the **Servers** tab in the bottom pane of JBDS. The JBoss EAP server should have been added in a previous lab. Right-click the server **Red Hat JBoss EAP 7.0 [Stopped]** and click on the green start button to start the server. Watch the **Console** until the server starts and you see the message:

```
INFO [org.jboss.as] (Controller Boot Thread) WFLYSRV0025: JBoss EAP 7.0.0.GA
(WildFly Core 2.1.2.Final-redhat-1) started
```

▶ **5.** Run the JUnit test and inspect the result.

5.1. Review the test class **EJBTest.java**

In the expanded **stateless-ejb** item in the **Project Explorer** tab in the left pane of JBDS, select **stateless-ejb > Java Resources > src/test/java > com.redhat.training.ejb** and double-click the **EJBTest.java** file.

```
...
@RunWith(Arquillian.class)
public class EJBTest {
 @Inject
 private Hello hello;

 @Deployment
 public static WebArchive createDeployment() {
 return ShrinkWrap.create(WebArchive.class, "stateless-ejb-
test.war").addClass(HelloBean.class).addClass(Hello.class)
 .addAsManifestResource(EmptyAsset.INSTANCE,
 ArchivePaths.create("beans.xml"));
 }

 @Test
 public void testHelloEJB() {
 hello.setName("John Doe");
 String result = hello.greet();
 assertEquals("Hello, John Doe", result);
 }
}
```

```
}
```

```
...
```

The test class is annotated with `@RunWith(Arquillian.class)` to ensure that Arquillian Runner is used by JBDS to deploy the application to the server for testing.

The `Hello` backing bean is injected with the following lines:

```
@Inject
private Hello hello;
```

- 5.2. Right-click the file name `EJBTest.java` on the left pane, click the **Run As** option and select **JUnit Test** to run the test method.

- 5.3. Expand the **JUnit** pane by double-clicking the **JUnit** tab.

Notice that the test failed due to a `NameNotFoundException`.

A `NameNotFoundException` is raised because the EJB class was never instantiated. To fix this, the `HelloBean` class needs to be annotated with a `@Stateless` annotation to make the class an EJB so that it can be injected and instantiated.

#### ► 6. Update `HelloBean` to be a stateless EJB.

- 6.1. Update `HelloBean` with the `@Stateless` annotation:

```
import javax.ejb.Stateless;

@Stateless
public class HelloBean {

 public String sayHello(String name) {
 // respond back with Hello, {name}.
 return "Hello, " + name;
 }
}
```

- 6.2. Save the changes by pressing `Ctrl+S`.

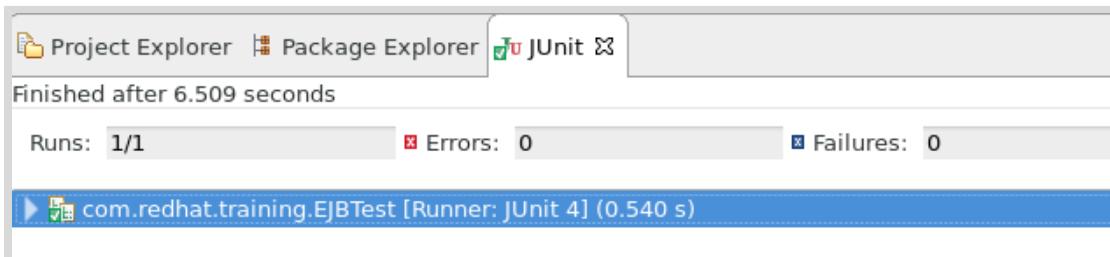
#### ► 7. Rerun the unit test and ensure that the tests pass.

- 7.1. To rerun the test, right-click the file name `EJBTest.java` on the left pane, click the **Run As** option and select **JUnit Test**.
- 7.2. Observe the server **Console** in JBDS. The following messages confirm JNDI bindings in EAP for the stateless session EJB

```
INFO [org.jboss.as.ejb3.deployment] (MSC service thread 1-3) WFLYEJB0473: JNDI
bindings for session bean named 'HelloBean' in deployment unit 'deployment
"test.war"' are as follows:
java:global/test/HelloBean!com.redhat.training.ejb.HelloBean
java:app/test/HelloBean!com.redhat.training.ejb.HelloBean
java:module/HelloBean!com.redhat.training.ejb.HelloBean
java:global/test/HelloBean
java:app/test/HelloBean
java:module/HelloBean
```

- 7.3. Observe the test result in the **JUnit Test** tab in JBDS.

This time the test is successful.



**Figure 3.5: JUnit Test Success.**

- 8. Deploy the Application on JBoss EAP using Maven by running the following commands:

```
[student@workstation ~]$ cd /home/student/JB183/labs/stateless-ejb
[student@workstation stateless-ejb]$ mvn wildfly:deploy
```

Once complete, you should see **BUILD SUCCESS** as shown in the following example:

```
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 17.116 s
[INFO] Finished at: 2016-12-01T07:26:55-05:00
[INFO] Final Memory: 35M/210M
[INFO] -----
```

Also validate the deployment in the server log shown in the **Console** tab in JBDS. The following should be in the log when the app is deployed successfully:

```
INFO [org.jboss.as.server] (management-handler-thread - 9) WFLYSRV0010: Deployed
"stateless-ejb.war" (runtime-name : "stateless-ejb.war")
```

- 9. Test the application in a browser.

- 9.1. Open the following URL in a browser on the workstation VM: <http://localhost:8080/stateless-ejb>.

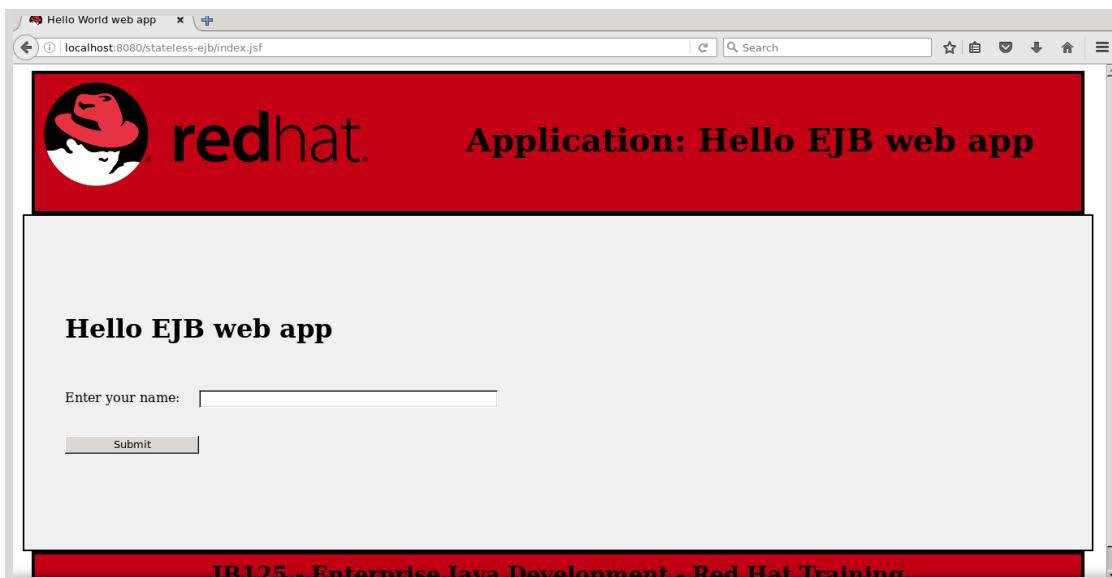


Figure 3.6: Application home page.

- 9.2. Enter **Shadowman** in the text box labeled **Enter your name:** and click **Submit**

The page updates with the message *Hello Shadowman*:

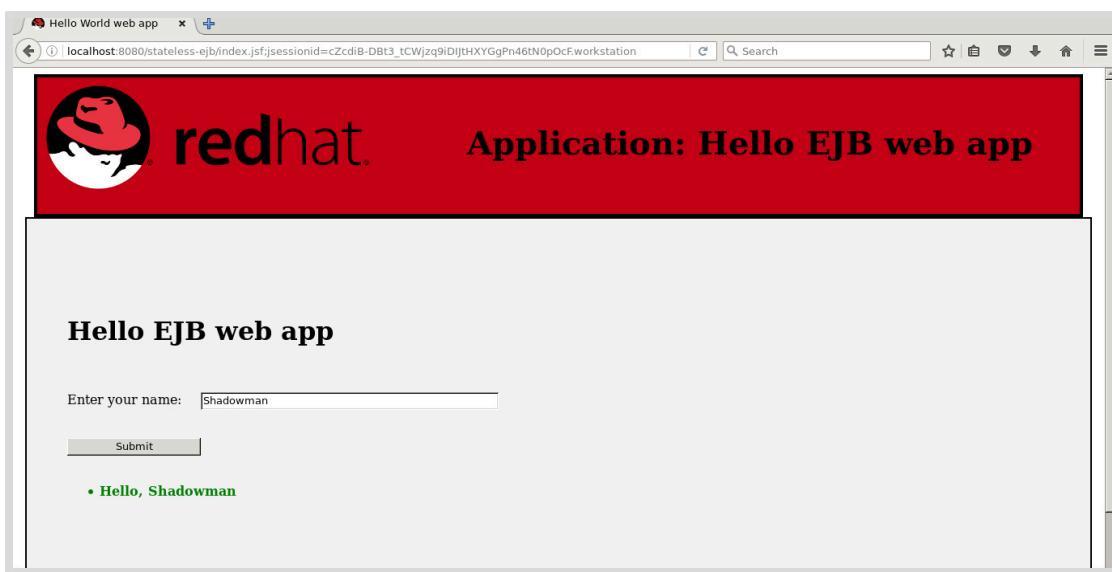


Figure 3.7: Application response.

► 10. Undeploy the application and stop EAP.

- 10.1. Run the following command to undeploy the application:

```
[student@workstation stateless-ejb]$ mvn wildfly:undeploy
```

- 10.2. Right-click on the **stateless-ejb** project in the **Project Explorer**, and select **Close Project** to close this project.

- 10.3. Right-click on the Red Hat JBoss EAP 7.0 server in the **JBDS Servers** tab and click **Stop**.

This concludes the guided exercise

# Accessing EJBs Locally and Remotely

## Objectives

After completing this section, students should be able to access an EJB both locally and remotely.

## Accessing EJBs

An EJB is a portable component containing business logic running on an application server. Clients can access EJBs either locally, if the client and EJB are part of the same application, or through a *Remote Interface*, if the EJB is running remotely.

If the client and EJB are local, that is, they are running within the same JVM process, the client can invoke all public methods in the EJB. In cases where the EJB is remote, a *Remote Interface*, which is a simple Java interface that exposes the business methods of an EJB, has to be provided. The EJB class implements the methods in the remote interface and its implementation details are hidden from the clients.

## Accessing Local EJBs Using the @EJB Annotation

Assume you have defined an EJB like the following:

```
@Stateless
public class TodoBean {

 public void addTodo(TodoItem item) {
 ...
 }

 public void findTodo(int id) {
 ...
 }
 ...
}
```

Clients can invoke methods on the EJB by injecting the EJB directly into code using the **@EJB** annotation:

```
public class TodoClient {

 @EJB
 TodoBean todo;

 TodoItem item = new TodoItem();
 item.setDescription("Buy milk");
 item.setStatus("PENDING");
```

```
//invoke EJB methods
todo.addTodo(item);
...
}
```

## Accessing Remote EJBs

In scenarios where the client is running outside the context of a Java EE application server, or if a Java EE component running on an application server needs to access another EJB that is deployed on a remote application server, you can use JNDI to lookup the EJB.

To ensure that an EJB can be consumed by remote clients, you have to declare an interface listing the business methods of the EJB, and have the EJB implement and override these methods. For example, assuming you wish to provide an EJB that does various mathematical operations, declare an interface and list the methods like the following:

```
package com.redhat.training.ejb;

public interface Calculator {
 public int add(int a, int b);
 public int multiply(int a, int b);
 ...
}
```

You now have to provide concrete implementations of these methods in the EJB and indicate that **Calculator** is the remote interface of the EJB by using the **@Remote** annotation:

```
package com.redhat.training.ejb;

@Stateless
@Remote(Calculator.class)
public class CalculatorBean implements Calculator {

 @Override
 public int add(int a, int b) {
 return a + b;
 }

 @Override
 public int multiply(int a, int b) {
 return a * b;
 }
 ...
}
...
```

Your EJB is now ready to be packaged and deployed on an application server and can serve remote clients.

## Looking Up Remote EJBs Using JNDI

The Java EE standards have specified a standard JNDI lookup scheme for clients to look up EJBs. It looks like the following:

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

- **application-name:** The application name is the name of the EAR that the EJB is deployed in (without the .ear extension). If the EJB JAR is not deployed in an EAR, then this is blank. The application name can also be specified in the EAR's `application.xml` deployment descriptor.
- **module-name:** By default, the module name is the name of the EJB JAR file (without the .jar suffix). The module name can be overridden in the `ejb-jar.xml` deployment descriptor.
- **bean-name:** The name of the EJB to be invoked (the implementation class).
- **fully-qualified-interface-name:** The fully-qualified class name of the remote interface. Include the full package name.

Considering the code listing above, assuming that the EJB is packaged inside a file called `calculator-ejb.jar`, which is further packaged into an EAR file called `myapp.ear`. Clients can look up the EJB using the following lookup string:

```
myapp/calculator-ejb/CalculatorBean!com.redhat.training.ejb.Calculator
```

When an EJB is deployed, the application server lists the different JNDI bindings for the EJB in the server logs. The following listing shows the JNDI entries if the EJB is packaged and deployed as a JAR file, and not as an EAR file:

```
INFO [org.jboss.as.ejb3.deployment] (MSC service thread 1-2) WFLYEJB0473: JNDI
bindings for session bean named 'CalculatorBean' in deployment unit 'deployment
"calculator-ejb.jar"' are as follows:

java:global/calculator-ejb/CalculatorBean!com.redhat.training.ejb.Calculator
java:app/calculator-ejb/CalculatorBean!com.redhat.training.ejb.Calculator
java:module/CalculatorBean!com.redhat.training.ejb.Calculator
java:global/calculator-ejb/CalculatorBean
java:app/calculator-ejb/CalculatorBean
java:module/CalculatorBean
```

A sample JNDI client program that uses the JNDI naming scheme to look up the remote EJB looks like the following:

```
package com.redhat.training.client;

public class CalculatorClient {

 public static void main(String[] args) throws Exception {

 String JNDI_URL= "myapp/calculator-ejb/CalculatorBean!com.redhat.training.ejb.Calculator";

 try {
 Context ic = new InitialContext();
 Calculator calc = (Calculator) ic.lookup(JNDI_URL);
```

```
System.out.println("Response from server = " + calc.add(1,2);
...
}
catch (Exception e) {
 // handle the exception
}

}
...
}
```

You also need to provide a file called `jndi.properties` in the class path of the client program, with the host name, IP address, port, and security details (if secured for remote access) of the remote application server where the EJB is running.

```
java.naming.factory.initial=org.jboss.naming.remote.client.InitialContextFactory
java.naming.provider.url=http-remoting://10.2.0.15:8080
jboss.naming.client.ejb.context=true
```

The JNDI API's `InitialContext` is a standard Java EE generic construct to do lookups of components deployed on an application server. It looks for a `jndi.properties` in the class path with a set of properties. Some of the properties are common to all application servers and some are specific to each application server.

Different application servers have their own specific ways to look up EJB components in an optimal manner, but this is out of scope for this course. Consult the References at the end of this section for more details.



## References

Further information is available in the EJB chapter of the *Development Guide* for Red Hat JBoss EAP:

<https://access.redhat.com/documentation/en/red-hat-jboss-enterprise-application-platform/7.0/>

## ► Guided Exercise

# Accessing an EJB Remotely

In this exercise you will create a command-line application to access a remote EJB deployed on JBoss EAP.

## Outcomes

You should be able to access an EJB deployed in a remote application server using JNDI lookup, and invoke its business methods.

## Before You Begin

Open a terminal window on the workstation VM and run the following command to download the files required for this lab.

```
[student@workstation ~]$ lab hello-remote setup
```

- 1. Open JBDS and import the Maven projects.

This guided exercise consists of two Maven subprojects, `hello-remote-ejb` and `hello-remote-client`, which are located under the main project folder in the `/home/student/JB183/labs/hello-remote` directory.

The `hello-remote-ejb` project installs a remotely accessible EJB in JBoss EAP so that it is available to external clients via JNDI lookups. The `hello-remote-client` project is a Java SE application that remotely accesses (from another JVM) the EJB.

- 1.1. Open JBDS by double-clicking the JBoss Developer Studio icon from the workstation desktop. Set the workspace as `/home/student/JB183/workspace` and click **OK**.

- 1.2. Import the `hello-remote-ejb` project by selecting from JBDS main menu **File > Import....**

From the **Import** dialog box, select **Maven > Existing Maven Projects** and click **Next >**. Click **Browse...** and choose `/home/student/JB183/labs/hello-remote/ejb` and click **OK** to choose the Maven project.

Click **Finish** to complete the import process in JBDS.

Once complete, a new project named `hello-remote-ejb` is listed in the **Project Explorer** view.

- 1.3. Import the `hello-remote-client` project by selecting from JBDS main menu **File > Import....**

From the **Import** dialog box, select **Maven > Existing Maven Projects** and click **Next >**. Click **Browse...** and choose `/home/student/JB183/labs/hello-remote/client` and click **OK** to choose the Maven project.

Click **Finish** to complete the import process in JBDS.

Once complete, a new project named `hello-remote-client` is listed in the **Project Explorer** view.

The `hello-remote-client` project is a Java SE application that remotely accesses (from another JVM) the EJB.

► 2. Explore the source code of the `hello-remote-ejb` project.

2.1. Review the `pom.xml` file.

Expand the `hello-remote-ejb` item in the **Project Explorer** tab in the left pane of JBDS, and double-click the `pom.xml` file. Click the `pom.xml` tab to view the full text of the `pom.xml` file.

Note the use of `maven-ejb-plugin` to package this EJB.

```
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-ejb-plugin</artifactId>
<version>${version.ejb.plugin}</version>
<configuration>
<ejbVersion>3.1</ejbVersion>
<generateClient>true</generateClient>
</configuration>
</plugin>
```

Note that the packaging has been declared as `ejb`. This tells Maven how to package the final deployable artifact.

```
<packaging>ejb</packaging>
```

2.2. Explore the business interface `HelloRemote.java` file.

In the **Project Explorer** tab in the left pane of JBDS, select `src/main/java > com.redhat.training.ejb` and double-click `HelloRemote.java` to view the file.

```
package com.redhat.training.ejb;

public interface HelloRemote {
 public String sayHello(String name);
}
```

Observe that this is a simple Java interface with a public method `sayHello` that takes a string `name` argument and returns a string. When working with EJBs, it is common to use interfaces to define the methods that are available, regardless of implementation.

2.3. Explore the implementation class `HelloBean.java` file.

In the **Project Explorer** tab in the left pane of JBDS, select `src/main/java > com.redhat.training.ejb` and double-click `HelloBean.java` to view the file.

```
@Stateless
public class HelloBean implements HelloRemote {
 public String sayHello(String name) {
 // respond back with "Hello, {name}".
 return "Hello, " + name;
 }
}
```

Observe that this EJB class implements the `sayHello` method of the `HelloRemote` interface and note the `@Stateless` annotation that marks this class as a stateless EJB.

- 3. Start EAP by selecting the **Servers** tab in JBDS. Right-click on the server entry **Red Hat JBoss EAP 7.0 [Stopped]** and click on the green Start option to start the server.

Watch the **Console** until the server starts and you see the started message:

```
INFO [org.jboss.as] (Controller Boot Thread) WFLYSRV0025: JBoss EAP 7.0.0.GA
(WildFly Core 2.1.2.Final-redhat-1) started
```

- 4. Build and deploy the EJB to JBoss EAP, and observe the JNDI bindings for the EJB.

- 4.1. Open a new terminal, and change to the `/home/student/JB183/labs/hello-remote/ejb/` folder.

```
[student@workstation ~]$ cd /home/student/JB183/labs/hello-remote/ejb
```

Build and deploy the EJB to JBoss EAP by running the following command:

```
[student@workstation ejb]$ mvn clean wildfly:deploy
```

The build should be successful and you should see the following output:

```
[student@workstation ejb]$ mvn clean wildfly:deploy
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building hello-remote-ejb 1.0
[INFO] -----
[INFO]
[INFO] <<< wildfly-maven-plugin:1.0.2.Final:deploy (default-cli) < package @
hello-remote-ejb <<<
...
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
```

- 4.2. Within JBDS, watch the **Console** tab and verify that the deployment is successful.

Observe the JNDI bindings for your stateless session bean:

```
JNDI bindings for session bean named 'HelloBean' in deployment unit 'deployment
"hello-ejb-remote.jar"' are as follows:
java:global/hello-ejb-remote/HelloBean!com.redhat.training.ejb.HelloRemote
java:app/hello-ejb-remote/HelloBean!com.redhat.training.ejb.HelloRemote
java:module/HelloBean!com.redhat.training.ejb.HelloRemote
java:global/hello-ejb-remote>HelloBean
java:app/hello-ejb-remote>HelloBean
java:module>HelloBean
```

JBoss EAP requires EJBs to be bound under the `java:jboss/exported/*` namespace to allow external clients to look up and invoke the EJB.

Observe that there are no "exported" JNDI bindings. You need to provide a remote interface to the EJB so that the EJB is bound under this namespace.

- ▶ 5. Edit the EJB project's implementation class `HelloBean.java` to enable remote JNDI lookups and redeploy the application.

- 5.1. Edit the implementation class `HelloBean.java` to enable remote JNDI lookups. Add the `@Remote` annotation to your implementation class and save the file.

```
import javax.ejb.Remote;
import javax.ejb.Stateless;

@Stateless
@Remote(HelloRemote.class)
public class HelloBean implements HelloRemote {

 public String sayHello(String name) {
 // respond back with "Hello, {name}".
 return "Hello, " + name;
 }
}
```

- 5.2. Save your changes by pressing `Ctrl+S`.

- 5.3. Run the following command to redeploy the EJB to EAP:

```
[student@workstation ejb]$ mvn clean wildfly:deploy
```

- 5.4. Observe the JNDI Bindings again. This time you should see the exported JNDI binding in the JBDS **Console** tab:

```
JNDI bindings for session bean named 'HelloBean' in deployment unit 'deployment
"hello-ejb-remote.jar"' are as follows:
java:global/hello-ejb-remote/HelloBean!com.redhat.training.ejb.HelloRemote
...
java:jboss/exported/hello-ejb-remote/HelloBean!
com.redhat.training.ejb.HelloRemote
...
```

- ▶ 6. Install the `hello-remote-ejb` artifact into the local repository using Maven so that it is available to the client project during compilation:

```
[student@workstation ejb]$ cd /home/student/JB183/labs/hello-remote/ejb
[student@workstation ejb]$ mvn install
```

The build should be successful and you should receive a successful build message.

```
[student@workstation ejb]$ mvn install
[INFO] -----
[INFO] Building hello-remote-ejb 1.0
[INFO] -----
...
[INFO] --- maven-install-plugin:2.4:install (default-install) @ hello-remote-ejb

[INFO] Installing /home/student/JB183/labs/hello-remote/ejb/target/hello-ejb-
remote.jar to /home/student/.m2/repository/com/redhat/training/hello-remote-
ejb/1.0/hello-remote-ejb-1.0.jar
[INFO] Installing /home/student/JB183/labs/hello-remote/ejb/pom.xml to /home/
student/.m2/repository/com/redhat/training/hello-remote-ejb/1.0/hello-remote-
ejb-1.0.pom
[INFO] Installing /home/student/JB183/labs/hello-remote/ejb/target/hello-ejb-
remote-client.jar to /home/student/.m2/repository/com/redhat/training/hello-
remote-ejb/1.0/hello-remote-ejb-1.0-client.jar
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
```

- 7. Install the parent POM into the local repository using Maven to make it available to the client project during compilation:

```
[student@workstation ejb]$ cd /home/student/JB183/labs
[student@workstation labs]$ mvn install
[INFO] -----
[INFO] Building JB125 Parent Project 1.0
[INFO] -----
...
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
```

- 8. Explore the **hello-client** project's source code, and update it to look up the **HelloBean** using JNDI.

- 8.1. Expand the **hello-client** item in the **Project Explorer** tab in the left pane of JBDS, and double-click the **pom.xml** file.

Click the **pom.xml** tab to view the full text of the **pom.xml** file.

Note the dependency on the **hello-remote-ejb** maven artifact that you installed previously.

```
<dependency>
<groupId>com.redhat.training</groupId>
<artifactId>hello-remote-ejb</artifactId>
<type>ejb-client</type>
<version>1.0</version>
</dependency>
```

The type of dependency is **ejb-client**. This tells Maven that this artifact is a client for the EJBs defined in the **hello-remote-ejb** artifact, which is used for code compilation.

#### 8.2. Explore the `HelloClient.java` file.

In Project Explorer tab in the left pane of JBDS, select `src/main/java > com.redhat.training.client` and double-click `HelloClient.java` to view the file.

```
public class HelloClient {
 public static void main(String[] args) throws Exception {

 //TODO Update JNDI address for the HelloBean
 String uriJNDI = "";
 try {
 Context ic = new InitialContext();
 // Lookup the remote EJB
 HelloRemote hello = (HelloRemote) ic.lookup(uriJNDI);
 // Invoke the sayHello method
 System.out.println("Response from server = " + hello.sayHello("Shadowman"));
 } catch (NamingException ex) {
 ex.printStackTrace();
 }
 }
}
```

#### 8.3. Fix the `HelloClient` code so that it can invoke the remote EJB by specifying a JNDI lookup string for the `InitialContext` instantiation.

Edit the `HelloClient.java` file and modify the `uriJNDI` variable to specify the JNDI lookup string to be used by the `InitialContext` instantiation. Set the JNDI name to match the following:

```
String uriJNDI = "hello-ejb-remote/HelloBean!com.redhat.training.ejb.HelloRemote";
```

#### 8.4. In Project Explorer tab in the left pane of JBDS, select `src/main/resources` and double-click `jndi.properties` to view the file in the main tab.

Click the **Source** tab to view and edit the properties file.

#### 8.5. Update the `jndi.properties` file to use `http-remoting` to access the EJB running on the local JBoss EAP server.

Set the `java.naming.provider.url` property to the value **`http-remoting://127.0.0.1:8080`** as shown in the following example:

```
java.naming.factory.initial=org.jboss.naming.remote.client.InitialContextFactory
java.naming.provider.url=http-remoting://127.0.0.1:8080
jboss.naming.client.ejb.context=true
```

The client is now connecting to the local JBoss EAP server instance running on port 8080.

Save your changes by pressing **Ctrl+S**.

- 9. Open a terminal window, and navigate to the `/home/student/JB183/labs/hello-remote/client` directory.

```
[student@workstation ~]$ cd /home/student/JB183/labs/hello-remote/client/
```

Run the client using Maven:

```
[student@workstation client]$ mvn package exec:java
```

You should see the following output:

```
[student@workstation client]$ mvn package exec:java
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building hello-client 1.0
[INFO] -----
...
[INFO] --- exec-maven-plugin:1.2.1:java (default-cli) @ hello-client ---
... Channel ID b334091e (outbound) of Remoting connection 63a791c4
to /127.0.0.1:8080
INFO: JBoss EJB Client version 2.1.6.Final-redhat-1 ...
Response from server = Hello, Shadowman
...
```

You will see the response **Hello, Shadowman** from the EJB. This verifies that the remote EJB has been invoked successfully.

- 10. Clean up the exercise and stop EAP.

10.1. Undeploy the EJB using the following Maven command:

```
[student@workstation ejb]$ mvn wildfly:undeploy
```

10.2. Right-click on the **hello-remote-ejb** project in the **Project Explorer**, and select **Close Project** to close this project.

10.3. Right-click on the **hello-client** project in the **Project Explorer**, and select **Close Project** to close this project.

10.4. Right-click on the Red Hat JBoss EAP 7.0 server in the **JBDS Servers** tab and click **Stop**.

This concludes the guided exercise.

# Describing the Life Cycle of an EJB

---

## Objective

After completing this section, students should be able to describe the life cycle of EJBs

## Understanding the EJB Life Cycle

EJB components in an application run within the context of a container inside an application server. The container is responsible for managing the *life cycle* (creation, execution and destruction) of the EJB's. Each of the different types of EJBs (Stateless, Stateful, Singleton, MDB) have their own life cycle.

## Describing the Stateful Session Bean Life Cycle

A stateful session bean has *three* distinct states in its life cycle:

- **Does Not Exist:** The stateful EJB is not created and does not exist in application server memory.
- **Ready:** The stateful EJB (object) is created in application server memory either by JNDI call or CDI injection and is ready to have its business methods invoked by clients.
- **Passivated:** Since a stateful EJB has object state that is persisted across multiple client calls, the application server may decide to *passivate* (deactivate) the EJB to secondary storage to optimize memory consumption. It will *activate* the EJB back into *Ready* state when a client invokes any method on the EJB. The developer does not have any direct control of activation and passivation and it is handled transparently by the application server based on certain algorithms.

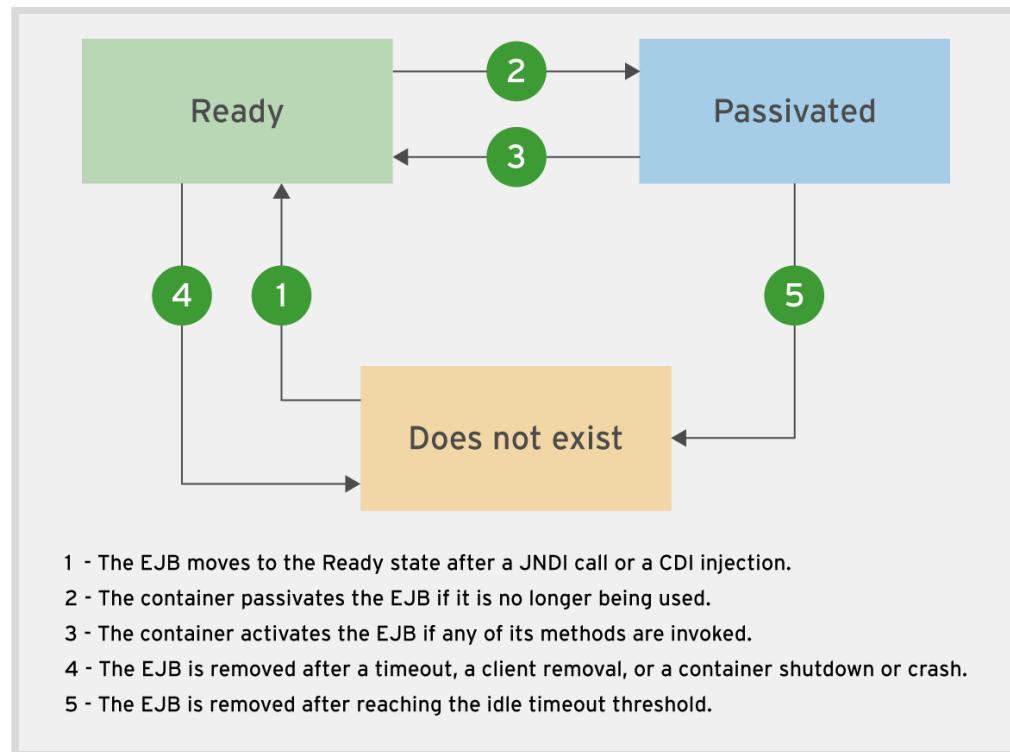


Figure 3.8: Reviewing the life cycle of a stateful session bean

## Describing the Stateless Session Bean Life Cycle

Since a stateless session bean has no state and is never passivated, it has just *two* distinct states in its life cycle:

- **Does Not Exist:** The stateless EJB is not created and does not exist in application server memory.
- **Ready:** The stateless EJB (object) is created in application server memory either by JNDI call or CDI injection and is ready to have its business methods invoked by clients.

Typically, the application server creates and maintains a pool of stateless session EJB instances in memory as a performance optimization. Whenever a client invokes business methods on the EJB, the application server allocates a bean from the pool. Once the method is executed, the bean is returned back to the pool. This process is transparent to the developer and is automatically handled by the application server. The size of the pool can be configured using deployment descriptors and in the application server configuration files.

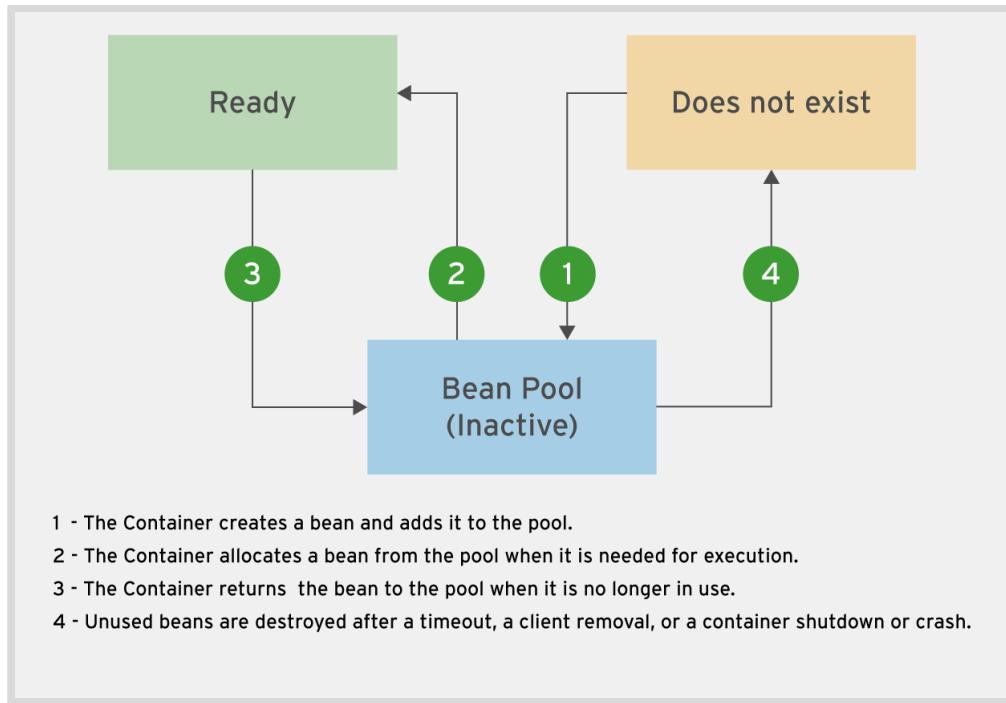


Figure 3.9: Reviewing the life cycle of a stateless session bean

## Describing the Singleton Session Bean Life Cycle

Similar to a stateless session bean, a singleton session bean has two distinct states in its life cycle:

- **Does Not Exist:** The singleton is not created and does not exist in application server memory.
- **Ready:** The singleton EJB (a single object) is created in application server memory at startup, or by CDI injection and is ready to have its business methods invoked by clients.

Since there is just one instance of the EJB for the duration of its life cycle, there is no concept of a pool. Concurrent access policies on the bean can be controlled by deployment descriptors or code level annotations.

## Reviewing the Bean Life Cycle Annotations

The Java EE standards for EJB specifies the concept of *Callbacks*. Callback is a mechanism by which the life cycle events of an enterprise bean can be intercepted and the developer can execute custom code during these events. The developer annotates methods in the bean with a set of annotations which are then invoked and executed by the application server.

The table below illustrates the different types of EJBs and the life cycle methods and annotations available for each:

Bean Type	Annotation	Description
Stateful Session Bean	@PostConstruct	method is invoked when a bean is created for the first time.
	@PreDestroy	method is invoked when a bean is destroyed.
	@PostActivate	method is invoked when a bean is loaded to be used after activation.

<b>Bean Type</b>	<b>Annotation</b>	<b>Description</b>
	@PrePassivate	method is invoked when a bean is about to be passivated.
Stateless Session Bean	@PostConstruct	method is invoked when a bean is created for the first time.
	@PreDestroy	method is invoked when a bean is removed from the bean pool or is destroyed.
Singleton Session Bean	@PostConstruct	method is invoked when a bean is created for the first time.
	@PreDestroy	method is invoked when a bean is destroyed.
	@Startup	The application server instantiates the singleton at startup.

## ► Quiz

# The Lifecycle of an EJB

Choose the correct answer(s) to the following questions:

- ▶ 1. **Which of the following two statements about the Stateful Session EJB life cycle are correct? (Choose two.)**
  - a. Clients can invoke methods on a stateful session bean that is in *Passivated* state.
  - b. Clients can invoke methods on a stateful session bean that is in *Ready* state.
  - c. The developer has direct control over stateful EJB passivation and activation by invoking methods directly on the bean.
  - d. The developer has no direct control over passivation or activation of stateful EJBs. The application server manages the activation and passivation of stateful EJBs.
  - e. Stateful EJBs in *Ready* state can be invoked only by remote clients through JNDI.
- ▶ 2. **Which of the following two statements about the Stateless Session EJB life cycle are correct? (Choose two.)**
  - a. The developer has direct control over stateless EJB passivation and activation.
  - b. The stateless EJB bean pool is of fixed size and cannot be changed.
  - c. The application server typically maintains a pool of stateless EJBs.
  - d. The size of stateless EJB bean pool can be changed by deployment descriptors and application server configuration files.
  - e. Stateless EJBs in *Ready* state can be invoked only by remote clients through JNDI.
- ▶ 3. **Which of the following statements about the Singleton EJB life cycle are correct?**
  - a. The application server maintains a pool of Singleton EJBs which are instantiated at startup.
  - b. Singleton EJBs can be passivated and activated just like Stateful EJBs.
  - c. The application server creates and maintains a single instance of the Singleton EJB at startup if the Singleton EJB is annotated with the **@Startup** annotation.
  - d. Pool size of the Singleton EJBs can be controlled by deployment descriptors and application server configuration files.

► 4. Which of the following three statements about the bean life cycle annotations are correct? (Choose three.)

- a. Methods on a stateful session EJB can be annotated with the **@PrePassivate** annotation.
- b. Methods on a stateless session EJB can be annotated with the **@PrePassivate** annotation.
- c. A Singleton EJB can be annotated with the **@PostConstruct** annotation.
- d. The **@Initialize** annotation on a Singleton EJB indicates that the application server will instantiate the Singleton at startup.
- e. Methods on a stateful session EJB can be annotated with the **@PreDestroy** annotation.
- f. Methods on a stateless session EJB can be annotated with the **@PostActivate** annotation.

## ► Solution

# The Lifecycle of an EJB

Choose the correct answer(s) to the following questions:

- ▶ 1. **Which of the following two statements about the Stateful Session EJB life cycle are correct? (Choose two.)**
  - a. Clients can invoke methods on a stateful session bean that is in *Passivated* state.
  - b. Clients can invoke methods on a stateful session bean that is in *Ready* state.
  - c. The developer has direct control over stateful EJB passivation and activation by invoking methods directly on the bean.
  - d. The developer has no direct control over passivation or activation of stateful EJBs. The application server manages the activation and passivation of stateful EJBs.
  - e. Stateful EJBs in *Ready* state can be invoked only by remote clients through JNDI.
  
- ▶ 2. **Which of the following two statements about the Stateless Session EJB life cycle are correct? (Choose two.)**
  - a. The developer has direct control over stateless EJB passivation and activation.
  - b. The stateless EJB bean pool is of fixed size and cannot be changed.
  - c. The application server typically maintains a pool of stateless EJBs.
  - d. The size of stateless EJB bean pool can be changed by deployment descriptors and application server configuration files.
  - e. Stateless EJBs in *Ready* state can be invoked only by remote clients through JNDI.
  
- ▶ 3. **Which of the following statements about the Singleton EJB life cycle are correct?**
  - a. The application server maintains a pool of Singleton EJBs which are instantiated at startup.
  - b. Singleton EJBs can be passivated and activated just like Stateful EJBs.
  - c. The application server creates and maintains a single instance of the Singleton EJB at startup if the Singleton EJB is annotated with the **@Startup** annotation.
  - d. Pool size of the Singleton EJBs can be controlled by deployment descriptors and application server configuration files.

► 4. Which of the following three statements about the bean life cycle annotations are correct? (Choose three.)

- a. Methods on a stateful session EJB can be annotated with the **@PrePassivate** annotation.
- b. Methods on a stateless session EJB can be annotated with the **@PrePassivate** annotation.
- c. A Singleton EJB can be annotated with the **@PostConstruct** annotation.
- d. The **@Initialize** annotation on a Singleton EJB indicates that the application server will instantiate the Singleton at startup.
- e. Methods on a stateful session EJB can be annotated with the **@PreDestroy** annotation.
- f. Methods on a stateless session EJB can be annotated with the **@PostActivate** annotation.

# Demarcating Implicit and Explicit Transactions

---

## Objectives

After completing this section, students should be able to describe implicit and explicit transactions.

## Understanding Transactions in Java EE Applications

A typical Java EE enterprise application stores and manipulates data in one or more persistent data stores like a relational database management system (RDBMS). Business-critical data stored in these databases is usually accessed concurrently by multiple applications and it is vitally important to ensure data integrity. *Transactions* ensure that data integrity is maintained by controlling concurrent access to the data, and to ensure that a failed business transaction does not leave the system in an inconsistent or invalid state.

A **Transaction** is a series of actions that must be executed as a single atomic unit. It follows an "all or nothing" approach, where either all the actions are executed successfully or, none are executed at all.

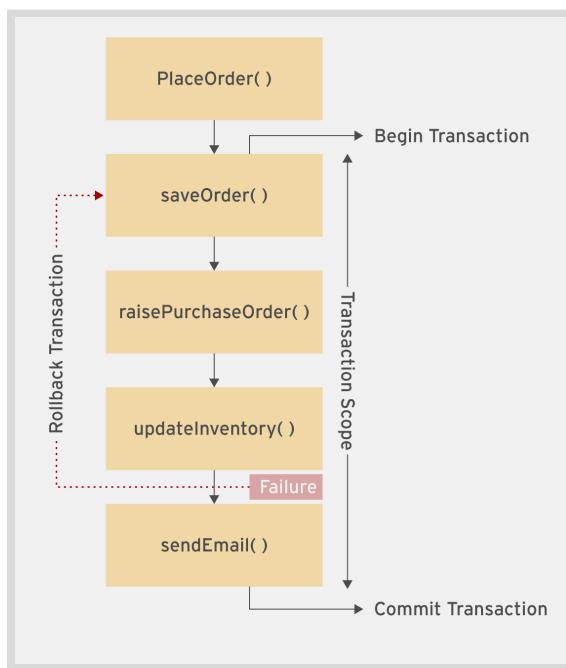


Figure 3.10: Using a transaction to rollback on failure

The figure above shows the steps for an order management application, where a sequence of operations has to be executed when a customer places an order. The transaction begins with the execution of the `saveOrder()` method which stores the order in an Orders database and then calls the `raisePurchaseOrder()` method, which raises a purchase order in another database maintained by the Finance department. The flow then moves to the `updateInventory()` method, which updates the inventory database and then sends an email to the customer using the `sendEmail()` method.

If all the methods in the transaction are executed without any errors or failures, the transaction is **Committed**. If, for example, there is a failure in the `updateInventory()` method, the application must ensure that the actions of the previous methods participating in the transaction (that is, `raisePurchaseOrder()` and `saveOrder()`) are reversed and that the overall state of the system reverts back to the state at the beginning of the transaction. This reversal is known as **Rollback**.

The Java EE standard specifies the *Java Transaction API (JTA)* that provides transaction management for applications running on a Java EE compliant application server. This API provides a convenient high-level interface for committing and rolling back transactions in applications. For example, if the Java Persistence API (JPA) is being used along with JTA, the developer does not have to explicitly write and keep track of SQL commits and rollback statements in the application. The JTA API takes care of these operations in a database independent manner.

There are two different ways to manage transactions in Java EE:

- **Implicit or Container Managed Transaction (CMT):** The application server manages the transaction boundary and automatically commits and rolls back transactions without the developer writing code for managing transactions. This is the default option unless explicitly overridden by the developer.
- **Explicit or Bean Managed Transaction (BMT):** The transactions are managed by the developer in code at the bean level (in EJBs). The developer is responsible for controlling the transaction scope and boundary explicitly.

## Reviewing Container Managed Transaction (CMT) Semantics

In CMT, the application server implicitly begins a transaction at the beginning of an EJB method and commits the transaction at the end of the method, unless there is an error, or an exception is thrown. In that case, the transaction is rolled back. Runtime exceptions automatically trigger a rollback by the application server. Nesting of transactions within a single bean method is not allowed. A developer can override the default transactional behavior at the method level with annotations called *Transaction Attributes*.

EJBs that use CMT must not use any JTA API methods that conflict with the application server's transaction scope and boundary. For example, JTA methods such as `commit()`, `setAutoCommit()`, and `rollback()` or JDBC classes like `java.sql.Connection` and the `commit()` and `rollback()` methods from `javax.jms.Session` cannot be used. If you require explicit control over transaction flow, you must use bean managed transactions. Also, EJBs that use CMT must not use the `javax.transaction.UserTransaction` interface.

## Setting Transaction Attributes

In CMT, *transaction attributes* control the scope of a transaction and allows a developer to declaratively manage transactions at the individual method level in an EJB.

For example, consider the following code snippet where one stateless EJB calls a method from another stateless EJB:

```
@Stateless
public class TodoService {

 @Inject
 UserService user;
```

```

public void login(String user, String password) {
 user.authenticate(user,password);
}
...
}

@Stateless
public class UserService {

 public boolean authenticate(String user, String password) {
 ...
}
...
}

```

**Note**

`@Inject` can inject any bean including EJBs, whereas `@EJB` can only inject EJBs. `@Inject` is described in more detail in the chapter titled Implementing Contexts and Dependency Injection.

Transaction attributes can be used to control the scope and context under which the `UserService` class methods are executed.

The Java EE specification defines six transaction attributes. They are illustrated below with reference to the code snippet above:

### **@TransactionAttribute(TransactionAttributeType.REQUIRED)**

If the `login()` method is running within a transaction and calls the `authenticate()` method in the `UserService` class, then the `authenticate()` executes within the same transaction. If there is no transaction when `authenticate()` is called, then the application server starts a new transaction before executing `authenticate()`. This is the default transaction attribute unless explicitly overridden with other transaction attribute annotations.

### **@TransactionAttribute(TransactionAttributeType.REQUIRES\_NEW)**

If the `login()` method is running within a transaction and calls the `authenticate()` method in the `UserService` class, then the application server suspends the transaction and starts a new transaction before executing `authenticate()`. Once `authenticate()` finishes executing, control moves back to `login()` and the suspended transaction resumes. If there is no transaction when `authenticate()` is called, then the application server starts a new transaction before executing `authenticate()`. This attribute ensures that your method **always** runs with a new transaction.

### **@TransactionAttribute(TransactionAttributeType.MANDATORY)**

If the `login()` method is running within a transaction and calls the `authenticate()` method in the `UserService` class, then the `authenticate()` executes within the same transaction. If there is no transaction when `authenticate()` is called, then the application server throws a `TransactionRequiredException`. Use the this attribute if you want a method to always execute in the transaction context of the calling client.

## **@TransactionAttribute(TransactionAttributeType.NOT\_SUPPORTED)**

If the `login()` method is running within a transaction and calls the `authenticate()` method in the `UserService` class, then the application server suspends the transaction and runs `authenticate()` without any transaction context. Once `authenticate()` finishes executing, control moves back to `login()` and the suspended transaction resumes. If there is no transaction when `authenticate()` is called, then the application server does *not* start a new transaction before executing `authenticate()`. Use this attribute for methods that don't need transactions.

## **@TransactionAttribute(TransactionAttributeType.SUPPORTS)**

If the `login()` method is running within a transaction and calls `authenticate()`, then `authenticate()` executes within the same transaction. If there is no transaction when `authenticate()` is called, then the application server does NOT start a new transaction before executing `authenticate()`.

## **@TransactionAttribute(TransactionAttributeType.NEVER)**

If the `login()` method is running within a transaction and calls the `authenticate()` method in the `UserService` class, then the application server throws a `RemoteException`. If there is no transaction when `authenticate()` is called, then the application server does *not* start a new transaction before executing `authenticate()`.

## **Setting Transaction Attributes**

Transaction attributes are declared by annotating the EJB class or method with a `javax.ejb.TransactionAttribute` annotation, and setting it to one of the `javax.ejb.TransactionAttributeType` enum constants. If you annotate the EJB with `@TransactionAttribute` at the class level, then the specified attribute is applicable to all methods in the EJB. Annotating specific methods with `@TransactionAttribute` applies the attribute to only that method. Method-level transaction attribute declarations override class-level declarations.

Consider the following EJB class with multiple methods:

```

@Stateless
public class TodoEJB {

 @TransactionAttribute(TransactionAttributeType.REQUIRED)
 public void createTodo(TodoItem item) {
 ...
 }

 @TransactionAttribute(TransactionAttributeType.NEVER)
 public List<TodoItem> listTodos() {
 ...
 }

 @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
 public void logCreateTodo(TodoItem item) {
 ...
 }
}

```

The `createTodo()` method is annotated with the `@TransactionAttribute(TransactionAttributeType.REQUIRED)`

transaction attribute, and the `listTodos()` method is annotated with the `@TransactionAttribute(TransactionAttributeType.NEVER)` transaction attribute, because this method simply does a read-only operation from the database and does not insert, delete, or update any data.

The `logCreateTodo()` method is annotated with the `@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)` transaction attribute, which ensures that this method is always executed with a new transaction, even if it is called from another method executing with its own transaction context.

## Understanding Bean Managed Transaction (BMT) Semantics

In scenarios where you need fine-grained control over when a transaction begins and ends, and to control when to commit and rollback, Bean Managed Transactions (BMT) can be used. You can use the `begin()`, `commit()`, and `rollback()` methods from the `javax.transaction.UserTransaction` interface to control the transaction boundaries and scope explicitly.

In applications using BMT, you should not use the `getRollbackOnly()` and `setRollbackOnly()` methods of the `javax.ejb.EJBContext` interface. Use the `getStatus()` and `rollback()` methods of the `javax.transaction.UserTransaction` interface instead.

A sample EJB with bean managed transaction looks like the following:

```
@Stateless①
@TransactionManagement(TransactionManagementType.BEAN)②
public class BeanManagedEJB {

 @Inject
 private UserTransaction tx;③

 public Integer saveOrder(Order order) {
 try {
 tx.begin();④

 Integer orderId = createOrder(order);
 raisePurchaseOrder(orderId);

 sendEmail("Your order # " + orderId + "has been placed
successfully...");

 tx.commit();⑤

 return orderId;
 } catch (Exception e) {
 tx.rollback();⑥
 return null;
 }
 ...
}
```

① This class is a stateless EJB.

- ❷ Mark this EJB as bean managed by using the `@TransactionManagement(TransactionManagementType.BEAN)` annotation.
- ❸ Inject a `UserTransaction` object. This is used for beginning, committing, and rolling back transactions in this EJB.
- ❹ Begin a transaction.
- ❺ If all the methods execute successfully without any errors, commit the transaction.
- ❻ If there is an exception due to some failure, perform a rollback of the transaction.

## ► Guided Exercise

# Demarcating Transactions

In this exercise, you will take a stateless EJB and update it to use bean-managed transactions.

### Outcomes

You should be able to implement a stateless EJB that uses bean managed transactions.

### Before You Begin

Open a terminal window on the **workstation** VM and run the following command to download the files required for this lab.

```
[student@workstation ~]$ lab bean-transactions setup
```

### Steps

- 1. Open JBDS and import the Maven project.
  - 1.1. Open JBDS by double-clicking the JBoss Developer Studio icon from the workstation desktop. Set the workspace as /home/student/JB183/workspace and click OK.
  - 1.2. In the JBDS menu, click **File > Import** to open the **Import** wizard.
  - 1.3. On the **Select** page, click **Maven > Existing Maven Projects**, and then click **Next**.
  - 1.4. In the **Maven projects** page, click **Browse** to open the **Select root folder** window. Navigate to the /home/student/JB183/labs/ directory. Select the **bean-transactions** folder, and then click **OK**.
  - 1.5. On the **Maven projects** page, click **Finish**.
  - 1.6. Watch the JBDS status bar to monitor the progress of the import operation. It may take a few minutes to download all the required dependencies.
- 2. Explore the application source code.
  - 2.1. Review the JSF page that calls the EJB by expanding the **bean-transactions** item in the **Project Explorer** tab in the left pane of JBDS. Further expand the **bean-transactions > src > main > webapp** folders and double-click the **index.xhtml** file.  
The Expression Language (EL) value `#{{hello.sayHello()}}`, is invoked upon web form submission.
  - 2.2. Review the request-scoped **Hello** backing bean used by the JSF page.  
Expand the **bean-transactions** item in the **Project Explorer** tab in the left pane of JBDS, select **bean-transactions > Java Resources > src/main/java > com.redhat.training.ui** and expand it. Double-click the **Hello.java** file.

Note that this EJB uses CDI to inject the PersonService EJB, which is where the transaction logic will be added.

### 2.3. Explore the PersonService.java EJB class.

In the expanded **bean-transactions** item in the **Project Explorer** tab in the left pane of JBDS, select **bean-transactions > Java Resources > src/main/java > com.redhat.training.service** and expand it. Double-click the **PersonService.java** file.

```
@Stateless
// TODO Add a transaction manage type of 'BEAN'

public class PersonService {

 @PersistenceContext
 private EntityManager entityManager;

 // TODO Inject a UserTransaction to be used by this EJB

 public String hello(String name) {
 try {
 // TODO start a new transaction

 // let's grab the current date and time on the server
 LocalDateTime today = LocalDateTime.now();

 // format it nicely for on-screen display
 DateTimeFormatter format = DateTimeFormatter.ofPattern("MMM dd yyyy hh:mm:ss
a");
 String fdate = today.format(format);

 // Create a new Person object and persist to database
 Person p = new Person();
 p.setName(name);
 entityManager.persist(p);

 // respond back with Hello and convert the name to UPPERCASE. Also, send the
 // current time on the server.
 return "Hello " + name.toUpperCase() + "!. " + "Time on the server is: " +
fdate;
 } catch(Exception e) {
 //TODO roll-back the transaction

 throw new EJBException(e);
 }
 }
}
```

Notice that this EJB is already marked as **@Stateless**, but currently does not contain any transaction management. The **hello()** method creates a new entry in the database for each person that enters a name into the UI and returns a greeting containing the current date and time.

- ▶ 3. Update PersonService to use bean managed transactions.

- 3.1. Annotate the class PersonService with the `@TransactionManagement(TransactionManagementType.BEAN)` annotation to enable bean managed transactions on the EJB:

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class PersonService {
 ...
}
```

This annotation prevents the container from managing the transactions and allows the EJB to manage the transactions manually.

- 3.2. Add the following code to use resource injection to inject an instance of the UserTransaction class into the EJB for manual transaction management:

```
public class PersonService {

 @PersistenceContext
 private EntityManager entityManager;

 //TODO Inject a UserTransaction to be used by this EJB
 @Resource
 UserTransaction tx;
 ...
}
```

The `@Resource` annotation tells the container to allocate a new transaction object and inject it into this EJB at runtime.

- 3.3. Add the following code to the EJB to begin the transaction:

```
...
public String hello(String name) {
 try {
 //TODO start a new transaction
 tx.begin();
 ...
}
```

Add the following code to commit the transaction:

```
...
public String hello(String name) {
 try {
 // TODO start a new transaction
 tx.begin();
 // let's grab the current date and time on the server
 LocalDateTime today = LocalDateTime.now();

 // format it nicely for on-screen display
 DateTimeFormatter format = DateTimeFormatter.ofPattern("MMM dd yyyy hh:mm:ss
a");
 String fdate = today.format(format);

 // Create a new Person object and persist to database
 Person p = new Person();
 }
}
```

```

p.setName(name);
entityManager.persist(p);

//TODO commit the transaction
tx.commit();

...

```

Add the following code to rollback the transaction in the event of an exception:

```

...
public String hello(String name) {
 try {
 // TODO start a new transaction
 tx.begin()
 // let's grab the current date and time on the server
 LocalDateTime today = LocalDateTime.now();

 // format it nicely for on-screen display
 DateTimeFormatter format = DateTimeFormatter.ofPattern("MMM dd yyyy hh:mm:ss
a");
 String fdate = today.format(format);

 // Create a new Person object and persist to database
 Person p = new Person();
 p.setName(name);
 entityManager.persist(p);

 //TODO commit the transaction
 tx.commit();
 // respond back with Hello and convert the name to UPPERCASE. Also, send the
 // current time on the server.
 return "Hello " + name.toUpperCase() + "!. " + "Time on the server is: " +
fdate;
 } catch(Exception e) {
 //TODO roll-back the transaction
 tx.rollback();
 throw new EJBException(e);
 }
}

...

```

Save the changes to the file using **Ctrl+S**.

- 3.4. After adding the transaction rollback, notice that JBDS informs you of an unhandled exception. Hover over the `tx.rollback()` line underlined in red and click **Add throws declaration**. The method header now looks like the following:

```

public String hello(String name) throws IllegalStateException, SecurityException,
SystemException {

```

Save the changes to the file using **Ctrl+S**.

- ▶ 4. Start EAP and deploy the application using Maven.

- 4.1. Click the **Servers** tab in the bottom pane of JBDS. Right-click the server **Red Hat JBoss EAP 7.0 [Stopped]** and click on the green start button to start the server. Watch the **Console** until you see the following message:

```
INFO [org.jboss.as] (Controller Boot Thread) WFLYSRV0025: JBoss EAP 7.0.0.GA
(WildFly Core 2.1.2.Final-redhat-1) started
```

- 4.2. Deploy the Application on JBoss EAP using Maven with the following commands:

```
[student@workstation ~]$ cd /home/student/JB183/labs/bean-transactions
[student@workstation bean-transactions]$ mvn wildfly:deploy
```

Once complete, you should see **BUILD SUCCESS**:

```
...
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 17.116 s
[INFO] Finished at: 2017-09-21T07:26:55-05:00
[INFO] Final Memory: 35M/210M
[INFO] -----
```

- 4.3. Validate the deployment in the server log shown in the **Console** tab in JBDS:

```
INFO [org.jboss.as.server] (management-handler-thread - 9) WFLYSRV0010: Deployed
"bean-transactions.war" (runtime-name : "bean-transactions.war")
```

► 5. Test the application.

- 5.1. In the **workstation** VM, use a browser to navigate to the following URL: <http://localhost:8080/bean-transactions>.
- 5.2. Enter **Shadowman** in the text box labeled **Enter your name:** and click **Submit**.  
The page updates with the message *Hello SHADOWMAN!. Time on the server is: Sep 21 2017 04:15:04 PM*

► 6. Undeploy the application and stop EAP.

- 6.1. Execute the following commands to undeploy the application:

```
[student@workstation ~]$ cd /home/student/JB183/labs/bean-transactions
[student@workstation bean-transactions]$ mvn wildfly:undeploy
```

- 6.2. Right-click on the **bean-transactions** project in the **Project Explorer**, and select **Close Project** to close this project.
- 6.3. Right-click on the **Red Hat JBoss EAP 7.0** server in the JBDS **Servers** tab and click **Stop**.

This concludes the guided exercise

## ► Lab

# Creating Enterprise Java Beans

In this lab you will convert a POJO class to an EJB in the To Do List Application, and display a list of todo items on a web page.

## Outcomes

You should be able to modify a prewritten To Do List Application and convert a POJO class to an EJB. The application accepts a list of things to do as input from the user, and displays the list of todo items on a web page.

## Before You Begin

Open a terminal window on the workstation VM and run the following command to download the files that are required for this lab.

```
[student@workstation ~]$ lab ejb-assessment setup
```

1. Open JBDS, and import the `ejb-assessment` project located in the `/home/student/JB183/labs/ejb-assessment` directory.
2. Review the `Item.java` class located in the `/home/student/JB183/labs/ejb-assessment/src/main/java/com/redhat/training/todo/model` folder.  
This class models a single todo item in the application. It has three attributes: an id, a description, and a boolean attribute indicating if the task is completed or not.
3. Review the `ItemRepository.java` class located in the `/home/student/JB183/labs/ejb-assessment/src/main/java/com/redhat/training/todo/data` folder.  
This class simulates an in-memory database and stores the list of todo items. It has methods to add items, view a single item, and view a list of all items.  
Observe that this class is annotated with `@ApplicationScoped`, which means that the objects of this class are kept in scope (alive) as long as the application is deployed and running on the application server.  
Observe that the `seedTodoList()` method has been annotated with `@PostConstruct`. This method populates the todo list with three items once the class has been initialized.
4. Explore the `ItemService.java` class, which is a simple POJO containing methods to add todo items, view a todo item, and list all todo items. This file is located in the `/home/student/JB183/labs/ejb-assessment/src/main/java/com/redhat/training/todo/service` folder.  
Observe that this class injects the `ItemRepository` class and invokes methods on it to add, view, and list all todo items.
5. Convert the `ItemService` POJO to a stateless EJB.
6. Explore the `ItemResourceRESTService` class, which provides REST end points for the front-end user interface (based on AngularJS). This file is located in the `/home/student/JB183/labs/ejb-assessment/src/main/java/com/redhat/training/todo/rest` folder.

Observe that this class needs to use the `ItemService` EJB to invoke the EJB's methods, and provides a JSON response to the front-end layer.

**7. Inject the `ItemService` EJB into the `ItemResourceRESTService` class.**

Add the following import into the `ItemResourceRESTService` class for the annotation:

```
import javax.ejb.EJB
```

**8. Start JBoss EAP from within JBDS.**

**9. Build and deploy the application to JBoss EAP using Maven.**

**10. Observe the `ItemService` EJB's JNDI Bindings in the JBoss EAP server logs:**

**11. Explore the application.**

11.1. Navigate to `http://localhost:8080/ejb-assessment` using a browser to access the application.

11.2. On the left side of the web page, observe that there are three todo items that were added in the `@PostConstruct` method of the `ItemRepository` class.

Observe the `Console` tab of JBDS and verify that logging information from the `register()` method in the `ItemService` EJB, and the `seedTodoList()` method in the `ItemRepository` class is visible.

11.3. On the right side of the web page, add a few todo items by entering some text in the `Description` field and clicking `Save`.

Verify that the task you added is seen in `Task List` table on the left.

Observe the `Console` tab of JBDS and verify that logging information from the `register()` method in the `ItemService` EJB is visible.

11.4. Add a few more tasks and observe that the application automatically displays a pagination bar in the bottom of the `Task List` table if there are more than five tasks.

**12. Open a new terminal window and run the following command to grade the lab:**

```
[student@workstation ~]$ lab ejb-assessment grade
```

The grading script should report `SUCCESS`. If there is a failure, check the errors and fix them until you see a `SUCCESS` message.

**13. Clean up.**

13.1. Undeploy the application from JBoss EAP using Maven with the following commands:

```
[student@workstation ~]$ cd /home/student/JB183/labs/ejb-assessment
[student@workstation ejb-assessment]$ mvn wildfly:undeploy
```

13.2. Right-click on the `ejb-assessment` project in the `Project Explorer`, and select `Close Project` to close this project.

13.3. Right-click on the Red Hat JBoss EAP 7.0 server in the JBDS `Servers` tab and click `Stop`.

This concludes the lab.

## ► Solution

# Creating Enterprise Java Beans

In this lab you will convert a POJO class to an EJB in the To Do List Application, and display a list of todo items on a web page.

## Outcomes

You should be able to modify a prewritten To Do List Application and convert a POJO class to an EJB. The application accepts a list of things to do as input from the user, and displays the list of todo items on a web page.

## Before You Begin

Open a terminal window on the workstation VM and run the following command to download the files that are required for this lab.

```
[student@workstation ~]$ lab ejb-assessment setup
```

1. Open JBDS, and import the `ejb-assessment` project located in the `/home/student/JB183/labs/ejb-assessment` directory.
  - 1.1. Open JBDS by double-clicking the JBoss Developer Studio icon from the workstation desktop. Leave the default configuration (`/home/student/JB183/workspace`) and click **OK**.
  - 1.2. In the JBDS menu, click **File > Import** to open the **Import** wizard.
  - 1.3. On the **Select** page, click **Maven > Existing Maven Projects**, and then click **Next**.
  - 1.4. In the **Maven projects** page, click **Browse** to open the **Select root folder** window. Navigate to the `/home/student/JB183/labs/` directory. Select the `ejb-assessment` folder, and then click **OK**.
  - 1.5. On the **Maven projects** page, click **Finish**.
  - 1.6. Watch the JBDS status bar to monitor the progress of the import operation. It may take a few minutes to download all the required dependencies.
2. Review the `Item.java` class located in the `/home/student/JB183/labs/ejb-assessment/src/main/java/com/redhat/training/todo/model` folder.  
This class models a single todo item in the application. It has three attributes: an id, a description, and a boolean attribute indicating if the task is completed or not.
3. Review the `ItemRepository.java` class located in the `/home/student/JB183/labs/ejb-assessment/src/main/java/com/redhat/training/todo/data` folder.  
This class simulates an in-memory database and stores the list of todo items. It has methods to add items, view a single item, and view a list of all items.  
Observe that this class is annotated with `@ApplicationScoped`, which means that the objects of this class are kept in scope (alive) as long as the application is deployed and running on the application server.

- Observe that the `seedTodoList()` method has been annotated with `@PostConstruct`. This method populates the todo list with three items once the class has been initialized.
- Explore the `ItemService.java` class, which is a simple POJO containing methods to add todo items, view a todo item, and list all todo items. This file is located in the `/home/student/JB183/labs/ejb-assessment/src/main/java/com/redhat/training/todo/service` folder.

Observe that this class injects the `ItemRepository` class and invokes methods on it to add, view, and list all todo items.

- Convert the `ItemService` POJO to a stateless EJB.

Annotate the `ItemService` class with the `@Stateless` annotation to convert this POJO to an EJB.

```
import javax.ejb.Stateless

@Stateless
public class ItemService {
```

- Explore the `ItemResourceRESTService` class, which provides REST end points for the front-end user interface (based on AngularJS). This file is located in the `/home/student/JB183/labs/ejb-assessment/src/main/java/com/redhat/training/todo/rest` folder.

Observe that this class needs to use the `ItemService` EJB to invoke the EJB's methods, and provides a JSON response to the front-end layer.

- Inject the `ItemService` EJB into the `ItemResourceRESTService` class.

Add the `@EJB` annotation to the `ItemService` declaration.

```
@EJB
ItemService itemService;
```

Add the following import into the `ItemResourceRESTService` class for the annotation:

```
import javax.ejb.EJB
```

- Start JBoss EAP from within JBDS.

Select the **Servers** tab in JBDS. Right-click on the server entry **Red Hat JBoss EAP 7.0 [Stopped]** and click on the green Start option to start the server. Watch the **Console** tab of JBDS until the server starts and you see the message:

```
INFO [org.jboss.as] (Controller Boot Thread) WFLYSRV0025: JBoss EAP 7.0.2.GA
(WildFly Core 2.1.8.Final-redhat-1) started
```

9. Build and deploy the application to JBoss EAP using Maven.

Open a new terminal, and change directory to the /home/student/JB183/labs/ejb-assessment folder.

```
[student@workstation ~]$ cd /home/student/JB183/labs/ejb-assessment
```

Build and deploy the EJB to JBoss EAP by running the following command:

```
[student@workstation ejb-assessment]$ mvn clean wildfly:deploy
```

The build should be successful and you should see the following output:

```
[student@workstation ejb-assessment]$ mvn clean package wildfly:deploy
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building TODO List Project 1.0
[INFO] -----
[INFO] [INFO] <<< wildfly-maven-plugin:1.0.2.Final:deploy (default-cli) < package
@ ejb-assessment <<<
...
[INFO] --- maven-war-plugin:2.1.1:war (default-war) @ ejb-assessment ---
...
[INFO] --- wildfly-maven-plugin:1.0.2.Final:deploy (default-cli) @ ejb-assessment
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
```

10. Observe the ItemService EJB's JNDI Bindings in the JBoss EAP server logs:

In the **Console** tab of JBDS, you should see the following:

```
JNDI bindings for session bean named 'ItemService' in deployment unit 'deployment
"ejb-assessment.war"' are as follows:
java:global/ejb-assessment/ItemService!
com.redhat.training.todo.service.ItemService
java:app/ejb-assessment/ItemService!com.redhat.training.todo.service.ItemService
java:module/ItemService!com.redhat.training.todo.service.ItemService
java:global/ejb-assessment/ItemService
java:app/ejb-assessment/ItemService
java:module/ItemService
```

11. Explore the application.

- 11.1. Navigate to <http://localhost:8080/ejb-assessment> using a browser to access the application.

- 11.2. On the left side of the web page, observe that there are three todo items that were added in the `@PostConstruct` method of the `ItemRepository` class.

Observe the **Console** tab of JBDS and verify that logging information from the `register()` method in the `ItemService` EJB, and the `seedTodoList()` method in the `ItemRepository` class is visible.

You should see the following logging output:

```
11:19:52,488 INFO [com.redhat.training.todo.service.ItemService] (default task-21)
 Fetching all TODO items...
11:19:52,489 INFO [com.redhat.training.todo.data.ItemRepository] (default task-21)
 Seeding TODO List cache...
```

- 11.3. On the right side of the web page, add a few todo items by entering some text in the **Description** field and clicking **Save**.  
Verify that the task you added is seen in **Task List** table on the left.  
Observe the **Console** tab of JBDS and verify that logging information from the **register()** method in the **ItemService** EJB is visible.  
For example, if you add a new task called **Plan Project**, you should see the following logging output:

```
11:21:23,985 INFO [com.redhat.training.todo.service.ItemService] (default task-23)
 Adding new task: Plan Project
```

- 11.4. Add a few more tasks and observe that the application automatically displays a pagination bar in the bottom of the **Task List** table if there are more than five tasks.
12. Open a new terminal window and run the following command to grade the lab:

```
[student@workstation ~]$ lab ejb-assessment grade
```

The grading script should report **SUCCESS**. If there is a failure, check the errors and fix them until you see a **SUCCESS** message.

13. Clean up.

- 13.1. Undeploy the application from JBoss EAP using Maven with the following commands:

```
[student@workstation ~]$ cd /home/student/JB183/labs/ejb-assessment
[student@workstation ejb-assessment]$ mvn wildfly:undeploy
```

- 13.2. Right-click on the **ejb-assessment** project in the **Project Explorer**, and select **Close Project** to close this project.
  - 13.3. Right-click on the Red Hat JBoss EAP 7.0 server in the **JBDS Servers** tab and click **Stop**.

This concludes the lab.

# Summary

---

In this chapter, you learned:

- An Enterprise Java Bean (EJB) is a portable Java EE component that is typically used to encapsulate business logic in an enterprise application. It runs on an application server and can be consumed by remote clients as well as other Java EE components running locally in the same JVM process.
- An EJB provides multi-threading, concurrency, transactions, and security for enterprise applications without requiring the developer to write code for these features explicitly. Furthermore, the developer can declaratively add annotations to the EJB to expose the business methods as web service end-points.
- There are two different types of EJB: *Session Beans* and *Message Driven Beans (MDB)*. Session beans can be of three types: *Stateless Session Beans (SLSB)*, *Stateful Session Beans (SFSB)* and, *Singleton Session Beans*.
- A *Message Driven Bean (MDB)* enables Java EE applications to process messages asynchronously. An MDB listens for JMS messages. For each message received, it performs an action. MDBs provide an event driven, loosely coupled model for application development.
- If an EJB client and the EJB are running locally in the same JVM process, clients can directly inject references to the EJB using the `@EJB` annotation. If the client is remote, then *JNDI lookups* are used.
- EJB components in an application run within the context of a container inside an application server. The container is responsible for managing the life cycle (creation, execution, and destruction) of the EJBs. Each of the different types of EJBs (Stateless, Stateful, Singleton, MDB) have their own life cycle.
- Java EE has support for *Transactions* to ensure that data integrity is maintained by controlling concurrent access to the data, and to ensure that a failed business transaction does not leave the system in an inconsistent or invalid state.
- In Java EE, transactions can be managed in two different ways: Container Managed Transactions (CMT) and Bean Managed Transactions (BMT).
- In CMT, the application server manages the transactions without the developer writing any explicit code and the scope can be controlled by using *Transaction attributes*. The application server can automatically perform a rollback when it encounters a failure or an exception.
- In BMT, the developer is responsible for managing the transactions and is in full control of the scope of transactions. The developer has to manually commit and rollback transactions if there is an exception or a failure.



## Chapter 4

# Managing Persistence

### Overview

**Goal** Create Persistence Entities with validations.

**Objectives**

- Describe the Persistence API.
- Persist data to a data store using entities.
- Annotate beans to validate data.
- Create a query using the Java Persistence Query Language.

**Sections**

- Describing the Persistence API (and Quiz)
- Persisting Data (and Guided Exercise)
- Annotating Classes to Validate Beans (and Guided Exercise)
- Creating Queries (and Guided Exercise)

**Lab** Managing Persistence

# Describing the Persistence API

---

## Objectives

After completing this section, students should be able to:

- Understand object relational mapping concepts.
- Describe entity class and annotations.
- Describe how to use EntityManager in an EJB.
- Describe a persistence context XML descriptor.
- Describe the impact of transactions on persistence.

## Object Relational Mapping

When an application stores data in a permanent store like a flat file, XML file, or a database for durability, it is known as persistence. Relational databases are one of the most common data stores an enterprise application uses to preserve data for reuse.

Business data in a Java EE enterprise application is defined as Java objects. These objects are preserved in corresponding database tables. Java objects and database tables use different data types, such as a `String` in Java and `Varchar` in a database, to store business data. As data moves between the application and the database as a result of write operations, it can cause differences between the object model and the relational model. This discrepancy is known as an impedance mismatch, and application developers must write code to account for it if the mismatch is not already handled by the persistence provider.

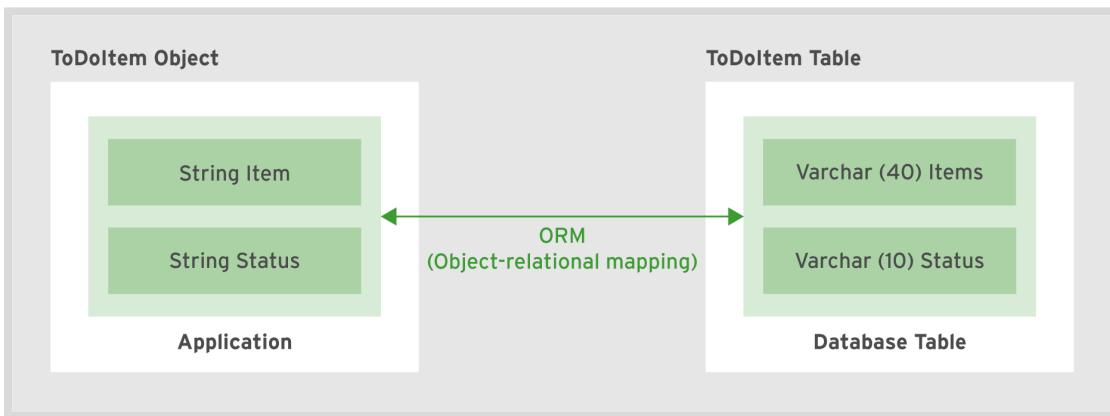


Figure 4.1: Impedance mismatch

The technique to automate bridging the impedance mismatch is known as Object Relational Mapping (ORM). ORM software uses metadata to describe mapping between the classes defined in an application and the schema of a database table. Mapping is provided in XML configuration files or annotations.

For example, you want to store `TodoItem` class objects in the `TodoItem` database table; ORM maps the Java class name to a database table name and the attributes in the class are mapped to the corresponding fields in the table automatically.

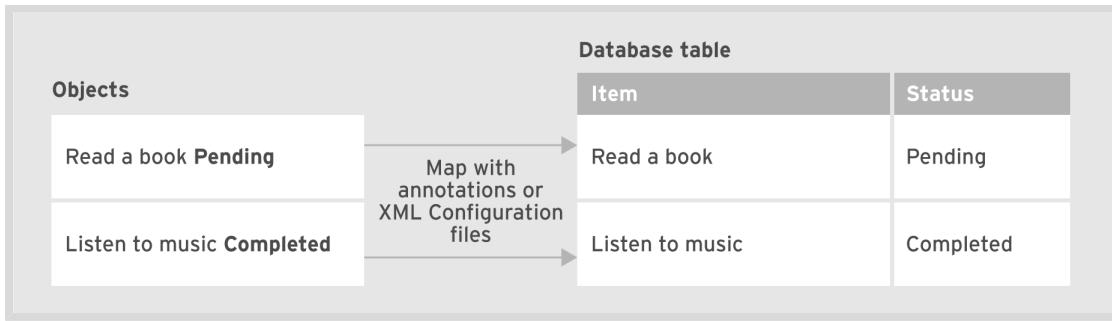


Figure 4.2: Object-relational mapping

Java EE provides the Java Persistence API (JSR 338) specification that is implemented by various ORM providers. There are many ORM software offerings available in the market, such as EclipseLink and Hibernate. A fully implemented ORM provides optimization techniques, caching, database portability, query language in addition to object persistence. The three key concepts related to the Java Persistence API are entities, persistence units, and persistence context.

## Entity Class and Annotations

An entity is a lightweight domain object that is persistable. An entity class is mapped to a table in a relational database. Each instance of an entity class has a primary key field. The primary key field is used to map an entity instance to a row in a database table. All non-transient attributes map to fields in a database table. In a database table, each persisted instance of an entity has a persistence identity that uniquely identifies it in a table. In Java, an entity is a Plain Old Java Object (POJO) class that is annotated with `@Entity` annotation. All the fields in an entity class are stored in the database by default and are known as persistent fields. The attributes that are declared as transient are not stored in a database table and are known as non-persistent.

## Declaring Entity Class

An entity class is declared as follows:

```

import javax.persistence.*;
import java.io.*;
@Entity
public class TodoItem implements Serializable {
 @Id
 private int id; //primary key -- required for an Entity class
 private String item;
 private String status;

 public TodoItem(){ } //No argument constructor

 // other constructor
 public TodoItem(String item,String status) {
 this.item=item;
 this.status=status;
 }

 //Setter and Getter methods
 public String getItem() {
 return item;
 }
}

```

```

public void setItem(String item) {
 this.item = item;
}

public String getStatus() {
 return status;
}

public void setStatus(String status) {
 this.status = status;
}
}

```

The default relationship between an entity class and a database table is:

### Default Entity to Table Mapping

Entity	Table
Entity class	Table name
Attributes of entity class	Columns in a database table
Entity instance	Record or row in a database table

## Using JPA Annotations

Annotations are used to decorate the Java classes, fields, and methods with metadata for mapping, configuration, queries, validation, and so on, that are compiled and made available at runtime. Here are some commonly used annotations:

### @Entity

The `@Entity` annotation specifies that a class is an entity. A Java class can be configured as an entity class without using an `@Entity` annotation by mapping it in the `orm.xml` configuration file. The `orm.xml` contains all configuration details required to declare a Java class as an entity.

### @Table

The `@Table` annotation is used to specify mapping between an entity class and a table. It is used when the name of an entity class is different from the name of a table in the database.

```

@Entity
@Table(name="ThingsToDo")
public class TodoItem {
 ...
}

```

The `TodoItem` entity class is mapped to the `ThingsToDo` table.

### @Column

The `@Column` annotation is used to map a field or property to a column in the database.

```

@Entity
@Table(name="ThingsToDo")
public class TodoItems implements Serializable {
 @Column(name="itemname")
 private String item;
 ...
}

```

An `item` attribute is mapped to the column `itemname` in the table.

## @Temporal

The `@Temporal` annotation is used with a `Date` type of attribute. Database stores a date in a different way than Java classes. `Temporal` annotation manages mapping for a `java.util.Date` or `java.util.Calendar` type and converts it to an appropriate date type in the database.

```

@Entity
public class TodoItem implements Serializable {
 ...
 @Temporal(TemporalType.DATE)
 private Date completionDate;
}

```

## @Transient

`Transient` annotation is used to specify a non-persistent field.

```

@Entity
public class TodoItem implements Serializable {
 ...
 @Transient
 private int countPending;
}

```

The `countPending` field is not saved to the database table.

## @Id

The `@Id` annotation is used to specify the primary key. The `id` field is used to identify a unique row in the database table.

```

@Entity
public class TodoItem implements Serializable {
 @Id
 private int id;
 ...
}

```

A primary key can be a simple Java type or a composite value, consisting of multiple fields. For a composite primary key, a primary key class is defined. `@EmbeddedId` or `@IdClass` annotation is used to specify the composite primary key.



## References

Further information about configuring composite keys is available in the *Public API* for Red Hat JBoss EAP 7, found at  
<https://developers.redhat.com/apidocs/eap/7.0.0/>

## ID Generation

Every entity instance is mapped to a row in a database table. Each row in a table is unique and is identified by a unique ID known as a persistent entity identity. The persistent entity identity is generated from the primary key field. A primary key field is required in every entity class. A simple primary key should be one of the following types:

- Java primitive types: byte, short, int, long, or char
- The java.lang.String type
- Java Wrapper classes for primitive types: Byte, Short, Integer, Long, or Character
- Temporal types: java.util.Date, or java.sql.Date

`@Id` annotation is used to specify a simple primary key. `@GeneratedValue` annotation is applied to the primary key field or property to specify the primary key generation strategy. `@GeneratedValue` annotation provides a `.GenerationType` element of the enum type. The four primary key generation strategies are as follows:

### GenerationType.AUTO

The AUTO strategy is the default ID generation strategy and means that the JPA provider uses any strategy of its choice to generate the primary key. Hibernate selects the generation strategy based on the database specific dialect.

```
@Entity
public class TodoItem implements Serializable {
 @Id
 @GeneratedValue(GenerationType.AUTO)
 private int id;
 ...
}
```

### GenerationType.SEQUENCE

The SEQUENCE strategy means that the JPA provider uses the database sequence to generate the primary key. The sequence must be created in the database, and the sequence name is provided in the `generator` element.

```
/* ITEMS_SEQ sequence
create sequence ITEMS_SEQ
MINVALUE 1
START WITH 1
INCREMENT BY 1
*/
```

```

@Entity
public class TodoItem implements Serializable {
 @Id
 @GeneratedValue(GenerationType.SEQUENCE, generator="ITEMS_SEQ")
 private int id;
 ...
}

```

## GenerationType.IDENTITY

The `IDENTITY` strategy means that the JPA provider uses the database identity column to generate the primary key.

```

@Entity
public class TodoItem {
 @Id
 @GeneratedValue(GenerationType.IDENTITY)
 private int id;
 ...
}

```

## GenerationType.TABLE

The `TABLE` strategy means that the JPA provider uses database ID generation table. This is a separate table that is used to generate the ID value. The ID generation table has two columns. The first column is a string that identifies the generator sequence, and the second column is an integer value that stores the ID sequence.

```

@Entity
public class TodoItem implements Serializable {
 @TableGenerator(name="Items_gen",
 table="ITEM_ID_GEN",
 pkColumnName="GEN_NAME",
 valueColumnName="GEN_VAL",
 pkColumnValue="ITEM_ID",
 allocationSize=60)
 @Id
 @GeneratedValue(Generator="Items_gen")
 private int id;
 ...
}

```

## Describing Entity Manager

The `EntityManager` API is defined to perform persistence operations. An entity manager obtains the reference to an entity and performs the actual CRUD (Create, Read, Update, and Delete) operations on the database. An `EntityManager` instance can be obtained from an `EntityManagerFactory` object. An entity manager works within a set of managed entity instances. These managed entity instances are known as the entity manager's *persistence context*. You can think of a persistence context as a unique instance of a persistence unit. A persistence

unit is a collection of all entity classes and a `persistence.xml` file stored in an application archive. The `persistence.xml` is a configuration file that contains information about the entity classes, data source, transaction type, and other configuration information.

## Creating Entity Manager in EJB

An `EntityManagerFactory` object is created for the persistence unit, and this object is used to obtain an instance of `EntityManager`.

```
@Stateless
public class ItemService {
 //ItemPU is the name of the persistence unit
 EntityManagerFactory emFactory =
 Persistence.createEntityManagerFactory("ItemPU");
 EntityManager em = emFactory.createEntityManager();

}
```

Another way to obtain an `EntityManager` instance in Java EE managed objects, such as an EJB, is the producer technique. An object can be injected using Context Dependency Injection (CDI). CDI is a set of component management services that allow type-safe dependency injection. CDI is discussed in greater detail later in this course. A producer class defines a producer method that returns the data type that is injected to another class.

```
public class EMProducer {
 @Produces
 @PersistenceContext(unitName= "ItemPU")
 private EntityManager em;
}
```

An EJB class can inject the `EntityManager` using `@Inject` annotation.

```
@Stateless
public class ItemService{
 @Inject
 private EntityManager em;

 public void registerItem(Item item) throws Exception {
 ...
 em.persist(item);
 ...
 }
 public void removeItem(Long id) throws Exception {
 ...
 em.remove(findById(id));
 ...
 }
 public void updateItem(Item item) {
 em.merge(item);
 }
}
```

## Describing Persistence Unit

A Persistence unit describes configuration settings related to a data source, transactions, concrete classes, and object-relational mapping. A Persistence unit is configured in a `persistence.xml` file in the application's META-INF directory. Every application that uses persistence has at least one persistence unit. A Persistence unit contains information about the persistence unit name, data source, and transactions type. The `persistence.xml` file is discussed more in the next section.

```
<?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="Items" transaction-type="JTA">
 <jta-data-source>java:jboss/datasources/MySQLDS</jta-data-source>
 <properties>
 <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect" />
 <property name="hibernate.hbm2ddl.auto" value="update" />
 <property name="hibernate.show_sql" value="true" />
 <property name="hibernate.format_sql" value="true" />
 </properties>
 </persistence-unit>
</persistence>
```

## Impact of Transactions on Persistence

A transaction is a single unit of work that is comprised of a series of operations. If any one operation fails in the transaction, then the entire transaction is rolled back to its original state prior to the start of the transaction. If all operations are able to execute, then the transaction is committed and no roll-back is required. When working with persistence, transactions ensure that changes to a database do not partially complete as a result of an operation failure. JPA provides transactional behavior for operations on JPA resources using two approaches for transactions:

- *Resource Local Transactions*
- *JTA Transactions*

Resource local transactions are transactions with a scope spanning a single resource, such as a data source. For example, if an application is configured to use resource local transactions, the entity manager that is tied to the non-JTA datasource uses the `EntityTransaction` class to manage the transaction. This transaction, however, only applies to operations on that individual datasource based on the entity manager, which limits more complex transactions that span multiple datasources or messaging systems.

In contrast, JTA transactions span all resources in a container. Rather than referring to the `EntityTransaction` from the entity manager, JTA uses the `UserTransaction` class that allows you to start, commit, or roll back transactions independent of the resource or resources. This separation of transaction from a single resource allows transactions to contain complex operations that span multiple resources, such as multiple data sources and JMS messaging systems.



## References

### Java Persistence JSR

<https://www.jcp.org/en/jsr/detail?id=338>



## References

Further information is available in the *Development Guide for Java Persistence API* for Red Hat JBoss EAP 7, found at

<https://docs.jboss.org/author/display/AS7/JPA+Reference+Guide/>

## ► Quiz

# Describing the Persistence API

Choose the correct answer(s) to the following questions:

- ▶ 1. Which of the following annotations is required to convert a Java SE class to an entity class?
  - a. @Table
  - b. @Produces
  - c. @Entity
  - d. @EntityManager
  
- ▶ 2. Which of the following properties are not defined in the persistence.xml file?
  - a. Persistence-unit name
  - b. Transaction Type
  - c. Datasource URL
  - d. Database provider
  - e. Provider specific parameters
  
- ▶ 3. Which of the following two statements are correct about the entity manager?  
(Choose two.)
  - a. Entity manager objects are mapped to the rows in a database table.
  - b. An entity manager performs actual Create, Read, Update and Delete (CRUD) operations on an entity.
  - c. An entity manager has a persistence context associated with it.
  - d. An entity manager can produce a collection of EntityManagerFactory objects.
  
- ▶ 4. Which ID generation strategy uses a database column to generate the ID?
  - a. SEQUENCE\_GENERATOR
  - b. TABLE
  - c. SEQUENCE
  - d. IDENTITY

## ► Solution

# Describing the Persistence API

Choose the correct answer(s) to the following questions:

- ▶ 1. Which of the following annotations is required to convert a Java SE class to an entity class?
  - a. @Table
  - b. @Produces
  - c. @Entity
  - d. @EntityManager
  
- ▶ 2. Which of the following properties are not defined in the persistence.xml file?
  - a. Persistence-unit name
  - b. Transaction Type
  - c. Datasource URL
  - d. Database provider
  - e. Provider specific parameters
  
- ▶ 3. Which of the following two statements are correct about the entity manager?  
**(Choose two.)**
  - a. Entity manager objects are mapped to the rows in a database table.
  - b. An entity manager performs actual Create, Read, Update and Delete (CRUD) operations on an entity.
  - c. An entity manager has a persistence context associated with it.
  - d. An entity manager can produce a collection of EntityManagerFactory objects.
  
- ▶ 4. Which ID generation strategy uses a database column to generate the ID?
  - a. SEQUENCE\_GENERATOR
  - b. TABLE
  - c. SEQUENCE
  - d. IDENTITY

# Persisting Data

## Objectives

After completing this section, students should be able to:

- Describe requirements for entity classes.
- Describe entity fields and properties.
- Describe the EntityManager interface and key methods.

## Creating an Entity Class

An entity class is similar to a standard POJO class, but an entity has several important distinctions that require management by the EntityManager. To convert a POJO class to an entity, prepend an `@Entity` annotation in the class header. In addition, each instance variable should be accessed through the use of getter and setter methods. Finally, the class must at least have one constructor that has no arguments, although the class can still have other constructors that take arguments.

The following is an example of an entity class:

```
@Entity
public abstract class Customer {

 @Id
 private int custId;
 private String custName;
 ...
 public Customer(){ } // No argument constructor

 //setter and getter methods
 public String getCustName() {
 return custName;
 }
 public void setCustName(String custName) {
 this.custName = custName;
 }
 ...
}
```

## Entity Fields and Properties

Non-transient data in an entity class is persisted to a database table. A JPA provider can both load data from a database table into an entity class, and store data from an entity class into a database table. The way the state is accessed by the provider is known as the *access mode*. There are two access modes: field-based access and property-based access.

## Field-based Access

Field-based access is provided by annotating fields. A persistent field in an entity class must be declared with private, protected, or package level access. A persistent field should be one of the following types:

- Java primitive types: `byte`, `short`, `int`, `long`, or `char`
- `java.lang.String` type
- Java Wrapper classes for primitive types: `Byte`, `Short`, `Integer`, `Long`, or `Character`
- Temporal types: `java.util.Date`, or `java.sql.Date`
- Enumerated types
- Embeddable classes, other entities, and collections of entities

The getter and setter methods may or may not be present. An example of field-based access is as follows:

```
@Entity
public class Customer implements Serializable {
 // Note that the fields are annotated
 @Id
 protected int custId;
 protected String custName;
 @Temporal(TemporalType.DATE)
 protected Date registrationDate;
 @Column(name="address")
 protected Address custAddress;
 ...
}
```



### Note

A `Serializable` interface is required for entity classes that are accessed through a remote interface.

Field based access provides additional flexibility because fields or helper methods that should not be part of the persistent state can be excluded using the `@Transient` annotation or by omitting the getter and setter methods.

## Property-based Access

To provide property-based access, getter and setter methods must be defined in a Java entity class. Property-based access provides better encapsulation, as the access is only through methods. Property-based access is provided by annotating the getter methods. The return type of the getter method determines the type of the property. The return type of the getter method must be the same as the type of an argument passed to the setter method. The getter and setter methods must be either public or protected, and must follow the Java bean's naming conventions. An example of property-based access is as follows:

```
@Entity
public class Customer implements Serializable {
 protected int custId;
```

```

protected String custName;
protected Date registrationDate;
protected Address custAddress;
...
//Note the getter methods are annotated
@Id
public int getCustId(){
 return custId;
}
public String getCustName(){
 return custName;
}
@Temporal(TemporalType.DATE)
public Date getRegistrationDate(){
 return registrationDate;
}
@Column(name="address")
public Address getCustAddress(){
 return custAddress;
}
...
//Setter methods
}

```

## Entity States

An entity can exist in one of four states during its lifetime. These four states are:

- **New State:** An entity instance created using Java's `new` operator is in a new or transient state. An entity instance does not have a persistent identity and is not yet associated with the persistence context.
- **Managed State:** An entity instance with a persistent identity and that is associated with a persistence context is in a managed or persistent state. When a change is made to the data in managed entity fields, it is synchronized with the database table data. An entity instance is in the managed state after an application calls the `persist`, `find`, or the `merge` method of an entity manager.
- **Removed State:** A persistent entity can be removed from the database table in many ways. A managed entity instance can be removed from the database table when a transaction is committed, or when a `remove` method of an entity manager is called. An entity is then in the removed state.
- **Detached State:** An entity has a persistent entity identity but is not associated with the persistence context. This can happen when the entity is serialized or at the end of a transaction. This state is known as a detached state of an entity.

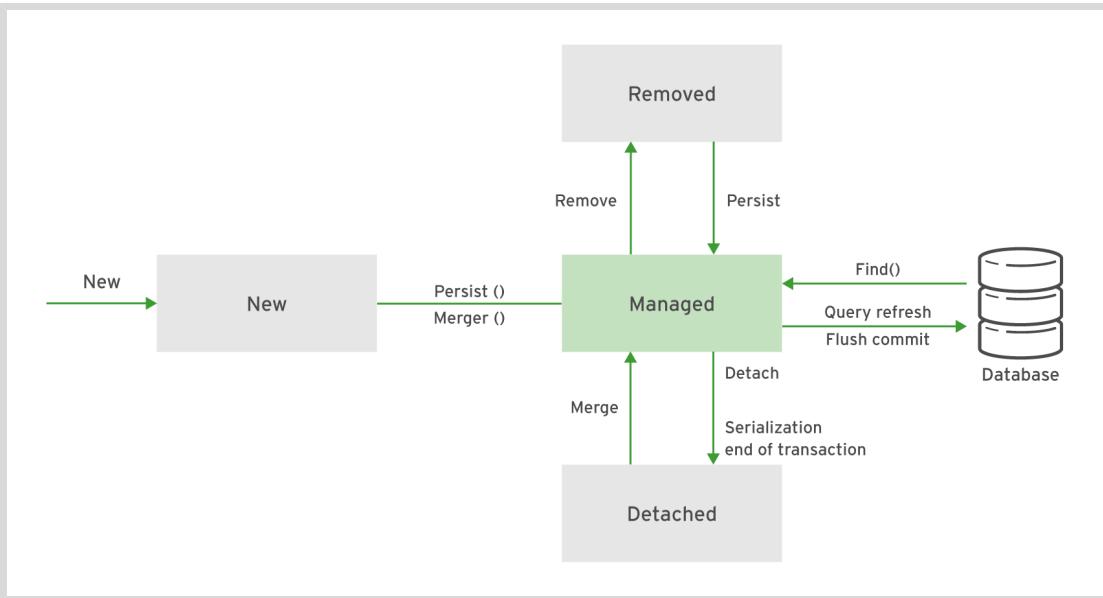


Figure 4.3: JPA components relationship

## EntityManager Interface and Key Methods

The `javax.persistence.EntityManager` interface is used to interact with the persistence context. The entity instances and their life cycles are managed within the persistence context. The `javax.persistence.EntityManager` API is used to create new entity instances, find entity instances by their primary key, query over entity instances, and remove the existing entity instances. The key methods of `EntityManager` are:

### **persist()**

The `persist()` method persists an entity and makes it managed. The `persist()` method inserts a row in a database table. The `persist()` method throws `PersistenceException` if persist operation fails.

```

@Stateless
public class CustomerServices {
 ...
 public void saveCustomer(Customer customer) {
 ...
 try{
 entityManager.persist(customer);
 }catch(PersistenceException persistenceException){
 // code to handle PersistenceException
 }
 }
}

```

### **find()**

The `find()` method searches an entity of a specific class by its primary key and returns a managed entity instance. If the object is not found, it returns a null.

```

@Stateless
public class CustomerServices {
 ...
 public void getCustomer(Customer customer) {
 ...
 Customer customer;
 try{
 customer = entityManager.find(Customer.class,custId);
 if (customer != null){
 System.out.print(customer.getCustName());
 } else {
 System.out.print("Not Found");
 }
 }catch(Exception exception){
 // code to handle PersistenceException
 }
 }
}

```

## contains()

The `contains()` method takes an instance as an argument and checks whether the instance is in the persistence context:

```

@Stateless
public class CustomerServices {
 ...
 public boolean saveCustomer(Customer customer) {
 ...
 entityManager.persist(customer);
 return entityManager.contains(customer);
 }
}

```

## merge()

The `merge()` method updates the data in a table for an existing detached entity. The `merge()` method inserts a new row in a database table for an entity that is in a new or a transient state. After the merge operation, an entity is in the managed state.

```

@Stateless
public class CustomerServices {
 ...
 public void updateCustomer(Customer customer) {
 ...
 Customer customer;
 try{
 customer = entityManager.find(Customer.class,custId);
 entityManager.merge(customer);
 }catch(Exception exception){
 // code to handle PersistenceException
 }
 }
}

```

```

 }
 }
}
```

## remove()

The `remove()` method deletes a managed entity. To delete a detached entity, call a `find()` method that returns a managed instance, and then call the `remove()` method.

```

@Stateless
public class CustomerServices {
 ...
 public void deleteCustomer(Customer customer) {
 ...
 Customer customer;
 try{
 customer = entityManager.find(Customer.class,custId);
 entityManager.remove(customer);
 }catch(Exception exception){
 // code to handle PersistenceException
 }
 }
}
```

## clear()

The `clear()` method clears the persistence context. After this operation, all managed entities are in the detached state.

```

...
try{
 entityManager.clear();
}catch(Exception exception){
 // code to handle PersistenceException
}
```

## refresh()

The `refresh()` method refreshes the state of an entity instance from a database table. The current data in an entity instance is overwritten by the data fetched from a database table.

```

...
try{
 entityManager.refresh(customer);
}catch(Exception exception){
 // code to handle PersistenceException
}
```

## Important Tags of persistence.xml File

The `persistence.xml` file is a standard configuration file that contains the persistence units. Each persistence unit has a unique name.

```
<?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="Items" ① transaction-type="JTA"> ②
 <jta-data-source>java:jboss/datasources/MySQLDS</jta-data-source>③
 <properties> ④
 <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect" />
 <property name="hibernate.hbm2ddl.auto" value="update" />
 <property name="hibernate.show_sql" value="true" />
 <property name="hibernate.format_sql" value="true" />
 </properties>
 </persistence-unit>
</persistence>
```

- ① `persistence-unit name` is the name of the persistence unit. The name of the persistence unit is used to obtain the `EntityManager`.
- ② `transaction-type` can be `JTA` or `RESOURCE_LOCAL`. Transaction type defines what type of transactions an application intends to perform. Container transactions use Java Transaction API (JTA), provided in every Java EE application server. In `JTA` type transactions, a container is responsible for creating and tracking the entity manager. In `RESOURCE_LOCAL`, you are responsible for creating and tracking the entity manager.
- ③ `jta-data-source` is the name of the data source. Each persistence unit must have a database connection. The JPA provider finds the data source by name with JNDI lookup service on startup.
- ④ Additional standard or vendor-specific properties can be set in the `properties` element. The `hibernate.Dialect` property specifies which database is used. The `hibernate.hbm2ddl.auto` property with an `update` value updates the schema automatically. The `hibernate.show-sql` property with a value as `true` enables logging of SQL statements to the console.

## Demonstration: Persisting Data

1. Run the following command to prepare files used by this demonstration.

```
[student@workstation ~]$ demo persist-data setup
```

2. Start JBDS and import the `persist-data` project.

This project is a simple JSF web app that displays all of the data stored in the database. It also uses CDI to inject the EJB responsible for querying the database.

3. Inspect the pom.xml file, and observe the dependency on hibernate libraries for the JPA specification and entity manager classes.

```
<dependency>
 <groupId>org.hibernate.javax.persistence</groupId>
 <artifactId>hibernate-jpa-2.1-api</artifactId>
 <scope>provided</scope>
</dependency>
<dependency>
 <groupId>org.hibernate</groupId>
 <artifactId>hibernate-entitymanager</artifactId>
 <scope>provided</scope>
</dependency>
```

4. Update the Member entity class to include the JPA annotations @Entity, @Id, and @GeneratedValue to convert the POJO class to an entity class.

```
@Entity
public class Member implements Serializable{
 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;

 private String emailId;
 ...
}
```

5. Update the saveMember method in the MemberService class to persist() the Member instance to the database.

```
public class MemberService {
 @PersistenceContext(unitName="member-unit")
 private EntityManager entityManager;

 public String saveMember(String emailId) {
 try {
 Member member = new Member();
 member.setEmailId(emailId);

 entityManager.persist(member);
 return "Congratulations ! you will get our weekly Tech Letter at " + emailId ;
 } catch (Exception e) {
 throw new EJBException(e);
 }
 }
}
```

6. Observe the MemberService class's getAllmembers() method. This method contains the JPA code that is used to query the database table.

```
public List<Member> getAllmembers() {
 TypedQuery<Member> query = entityManager.createQuery("SELECT m FROM Member m",
 Member.class);
 List<Member> members = query.getResultList();
 return members;
}
```

7. Start the local JBoss EAP server inside JBDS.

In the **Servers** tab in the bottom pane of JBDS, right-click the server **Red Hat JBoss EAP 7.0 [Stopped]** and click on the green button to start the server. Watch the **Console** until the server starts and you see the started message:

```
INFO [org.jboss.as] (Controller Boot Thread) WFLYSRV0025: JBoss EAP 7.0.0.GA
(WildFly Core 2.1.2.Final-redhat-1) started
```

8. Deploy the application to the local JBoss EAP server, and test it in a browser by running the following commands in a terminal window:

```
[student@workstation ~]$ cd /home/student/JB183/labs/persist-data
[student@workstation persist-data]$ mvn wildfly:deploy
```

Open <http://localhost:8080/persist-data/> in a browser on the workstation VM and make sure that the application is properly storing and displaying the members' email addresses.

9. Undeploy the application and stop the server.

```
[student@workstation persist-data]$ mvn wildfly:undeploy
```



## References

Further information is available in the *Development Guide for Java Persistence API* for Red Hat JBoss EAP 7, found at  
<https://docs.jboss.org/author/display/AS7/JPA+Reference+Guide/>

## ► Guided Exercise

# Persisting Data

In this exercise, you will persist application data to the database.

## Outcomes

You should be able to create an entity class and persist entity data.

## Before You Begin

Open a terminal window on the `workstation` VM and run the following command to download the lab files required for this guided exercise.

```
[student@workstation ~]$ lab persist-entity setup
```

## Steps

- 1. Import the `persist-entity` project into the JBoss Developer Studio IDE (JBDS).
  - 1.1. Start JBDS by double-clicking the JBDS icon on the `workstation` VM desktop.
  - 1.2. In the Eclipse Launcher window, enter `/home/student/JB183/workspace` in the **Workspace** field, and then click **Launch**.
  - 1.3. In the JBDS menu, click **File > Import** to open the **Import** wizard.
  - 1.4. On the **Select** page, click **Maven > Existing Maven Projects**, and then click **Next**.
  - 1.5. In the **Maven projects** page, click **Browse** to open the **Select root folder** window. Navigate to the `/home/student/JB183/labs/` directory. Select the `persist-entity` folder, and then click **OK**.
  - 1.6. On the **Maven projects** page, click **Finish**.
  - 1.7. Watch the JBDS status bar to monitor the progress of the import operation. It may take a few minutes to download all the required dependencies.



### Note

The `persist-entity` project has a compilation error when it is imported. This error is resolved in the lab steps that follow.

- 2. Convert a Simple Java Person class to an entity class.
  - 2.1. In the Project Explorer tab in the left pane of JBDS, select `persist-entity > Java Resources > src/main/java > com.redhat.training.model` and expand the package.
  - 2.2. Double-click the `Person.java` file in the selected `com.redhat.training.model` package to open the class in the editor.

- 2.3. Add the `@Entity` annotation to the `Person` class in the `com.redhat.training.model` package. Add the `@Entity` annotation and import the `javax.persistence.Entity` library.

```
//add the required libraries
import javax.persistence.Entity;
//add @Entity annotation to the Person class
@Entity
public class Person {

}
```

**Note**

Adding the `@Entity` annotation creates a compilation error. This error can be safely ignored and will be resolved in a later step.

- 2.4. The `Person` entity class must implement `Serializable` interface. Import and implement the `Serializable` interface.

```
import javax.persistence.Entity;
import java.io.Serializable;

@Entity
public class Person implements Serializable {

}
```

- 2.5. Add the `@Id` and `@GeneratedValue(strategy = GenerationType.IDENTITY)` annotations to the `id` attribute of the `Person` class to make it a primary key with key generation strategy as `IDENTITY`. Add the `@Column(name="name")` annotation to the `personName` attribute to map it to the `name` field in the database table. Import the required libraries.

```
...
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Column;

@Entity
public class Person implements Serializable {
 //add annotations for primary key
 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;
 //add @Column(name="name") annotation to map column in database table
 @Column(name="name")
 private String personName;

}
```

- 2.6. Save your changes by pressing **Ctrl+S**.
- 3. Open PersonService class in the `com.redhat.training.services` package and add the persistence functionality to save Person to the database and to find a person from the database.
- 3.1. In the Project Explorer tab in the left pane of JBDS, select `persist-entity > Java Resources > src/main/java > com.redhat.training.services`, and expand the package.
  - 3.2. Double-click the `PersonService.java` file in the `com.redhat.training.services` package to open the `PersonService` class in the editor pane.
  - 3.3. The `EntityManager` object is required to perform the persistence operations in the `PersonService` class. Add a `@PersistenceContext` annotation to get an `EntityManager` object:

```
import javax.persistence.PersistenceContext;
import javax.persistence.EntityManager;
@Stateless
public class PersonService {
 //TODO: obtain an EntityManager instance using @PersistenceContext
 @PersistenceContext(unitName="hello")
 private EntityManager entityManager;
 ...
}
```

- 3.4. Persist a Person into the database using the entity manager, add the following code to the `public String hello(String name)` method as follows:

```
public String hello(String name) {
 ...
 // call persist() method of entity manager to save the data
 entityManager.persist(p);
 ...
}
```

- 3.5. Find the name of a person using `id`, add the method `getPerson(Long id)` to the `PersonService` class. In the return statement, use the entity manager's `find()` method to return the Person's name attribute based on the `id`.

```
// TODO:add public String getPerson(Long id) method here to fetch result
//by Person id using find() method

public String getPerson(Long id) {
 return entityManager.find(Person.class, id).getPersonName();
}
```

- 3.6. Observe the `getAllPersons()` method that returns all of the Person objects stored in the database:

```
// Get all Person objects in the Database
public List<Person> getAllPersons() {
 TypedQuery<Person> query = entityManager.createQuery("SELECT p FROM Person p",
 Person.class);
 List<Person> persons = query.getResultList();

 return persons;
}
```

3.7. Save your changes by pressing **Ctrl+S**.

- 4. Open the **Hello** class in the **com.redhat.training.ui** package. Uncomment both the **getPerson()** and **getPersons()** methods to add the front end functionality to view the name of a single person and all of the names stored in the database.
- 4.1. In the **Project Explorer** tab in the left pane of JBDS, select **persist-entity > Java Resources > src/main/java > com.redhat.training.ui**, and expand the package.
  - 4.2. Double-click the **Hello.java** file in the selected **com.redhat.training.ui** package. The **Hello** class opens in the editor pane.
  - 4.3. Remove the comments on the **public void getPerson()** method.

```
public void getPerson() {
 try {
 String response = personService.getPerson(id);
 FacesContext.getCurrentInstance().addMessage(null, new FacesMessage(response));
 }catch(Exception e){
 System.out.println(e.getCause());
 if(e.getCause() != null && e.getCause() instanceof
 ConstraintViolationException) {
 ConstraintViolationException ex = (ConstraintViolationException) e.getCause();
 String violations = "";
 for(ConstraintViolation<?> cv: ex.getConstraintViolations()) {

 violations += cv.getMessage() + "\n";

 System.out.println("Violations: "+violations);
 }
 FacesContext.getCurrentInstance().addMessage(null, new
 FacesMessage(violations));
 }
 }
}
```

4.4. Remove the comments on **public List<Person> getPersons()** method.

```
public List<Person> getPersons() {
 return personService.getAllPersons();
}
```

- 5. Build and deploy the application.

- 5.1. Start EAP by selecting the Servers tab in the bottom pane of JBDS. Right-click the server **Red Hat JBoss EAP 7.0 [Stopped]** and click on the green start button to start the server.
- 5.2. Build the persist-entity application using the following commands in the terminal window:

```
[student@workstation ~]$ cd /home/student/JB183/labs/persist-entity
[student@workstation persist-entity]$ mvn clean package
```

- 5.3. Deploy the persist-entity application using the following command in the terminal window:

```
[student@workstation persist-entity]$ mvn wildfly:deploy
```

► 6. Test the application for persistence.

- 6.1. Use a web browser on the **workstation** VM to navigate to `http://localhost:8080/persist-entity/` to access the persist-entity application.



Figure 4.4: The persist-entity application

- 6.2. Enter **Samuel** in the **Enter your name** field and click **Submit**.
- 6.3. Verify that the server processes the input and responds with a greeting using the name you entered as well as the current time on the server.

The screenshot shows a web application titled "Persist Entity web app". At the top, there is a form with a label "Enter your name:" followed by an input field containing "Samuel". Below the input field is a "Submit" button. Underneath the form, a green bullet point says "Hello SAMUEL! Time on the server is: Oct 23 2017 05:49:30 AM". At the bottom of the page, there are two links: "View all names" and "Search a name".

Figure 4.5: The persist-entity application response

- 6.4. Repeat the previous step at least 2 more times, entering different names to populate the database.
- 6.5. Test that all the names are stored in the database by clicking [View all names](#).

The screenshot shows a web application titled "These people said hello". At the top, it says "Persons:". Below that is a bulleted list of names: "Name: Samuel", "Name: Gregg", "Name: Tim", "Name: John", and "Name: David". At the bottom left, there is a link "Back".

Figure 4.6: The list of all names

- Click the Back [/persist-entity/index.xhtml] link to go to the home page.
- 6.6. To find an individual person's name in the database, click [Search a name](#) link.

## Persist Entity web app

Enter Id:

[Back](#)

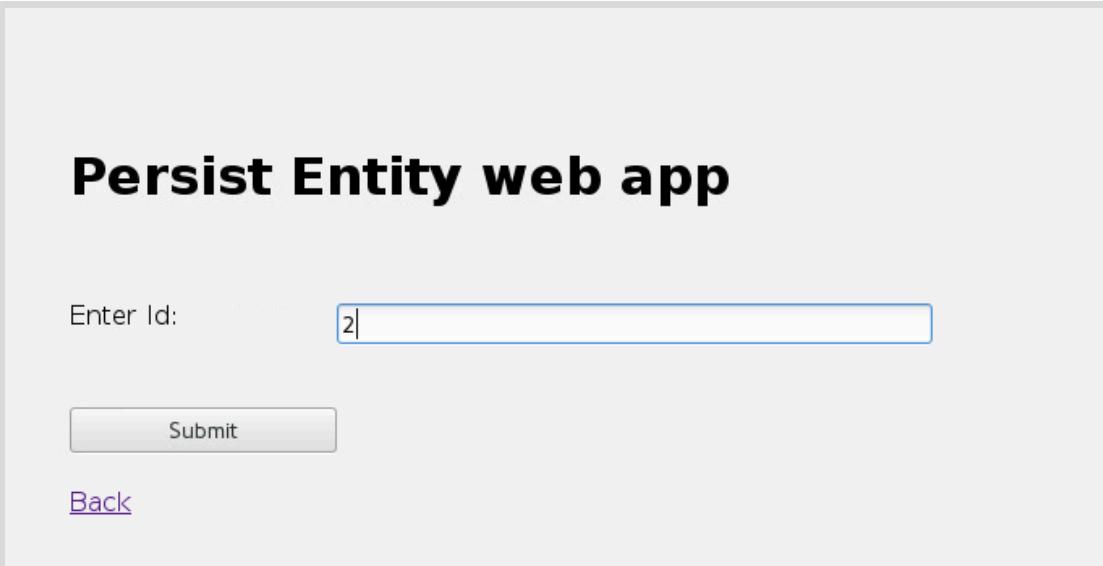


Figure 4.7: Finding an individual name in the database

- 6.7. Enter the value for an **id** in the **Enter Id:** field and click **Submit**.

## Persist Entity web app

Enter Id:

• Gregg

[Back](#)



Figure 4.8: Display of an individual name from the database

Click **Back** to go to the home page.

- 7. Undeploy the application and stop EAP.

- 7.1. In the terminal window where you ran the Maven command to deploy the application, run the following command to undeploy the application:

```
[student@workstation persist-entity]$ mvn wildfly:undeploy
```

- 7.2. Right-click on the **persist-entity** project in the **Project Explorer**, and select **Close Project** to close this project.

- 7.3. Right-click **Red Hat JBoss EAP 7.0** in the **Servers** tab and then click **Stop** to stop the EAP instance.

This concludes the guided exercise.

# Annotating Classes to Validate Beans

---

## Objectives

After completing this section, students should be able to:

- Explain bean validation.
- Use bean validation constraints and annotations.
- Compare automatic and manual invocation.

## Bean Validation

Java applications store data in Java objects. These Java objects travel over the network, passed as arguments to methods and are present in the different layers of a Java EE application. To maintain data integrity, data validation is a major requirement in application logic. Developers need to write data validation code in the different layers of the application for data validation, which is error-prone and time consuming. The *bean validation* API specification is provided to avoid code duplication and to simplify data validation. Bean validation is a model for validating data in Java objects by using built-in and custom annotations that can apply predefined constraints. Bean validation is common to all layers of Java EE and Java web applications. Java provides bean validation 1.1 API in JSR 349. JPA supports runtime validation of entity classes through the bean validation API. JBoss EAP is fully compliant with JSR 349.

## Bean Validation Constraints and Annotations

Validation constraints are the rules that are applied to validate data. These constraints are applied in the form of annotations to attributes, methods, properties, or constructors. Bean validation 1.1 permits the use of validation constraints on arguments and return values of methods and constructors. Java provides built-in constraints, and also supports custom constraints defined by the user. The `javax.validation.constraints` package contains several built-in constraints. Some common annotations:

Annotation	Description	Example
<code>@NotNull</code>	<code>@NotNull</code> annotation verifies that the value in the field or property is not null.	<code>@NotNull private String itemName;</code>
<code>@Null</code>	<code>@Null</code> annotation verifies that the value in the field or property is null.	<code>@Null private String comments;</code>

Annotation	Description	Example
@Size	@Size annotation verifies that the size of the field is between the min and max including the boundary values.	@Size (min=3, max=40) private String name;
@Min	@Min annotation verifies that the value in the field or property is greater than or equal to the value defined in the Min. The value is a required element of the long type.	@Min(100) private int minStock;
@Max	@Max annotation verifies that the value in the field or property is lesser than or equal to the value defined in the Max. The value is a required element of the long type.	@Max(1000) private int maxStock;

Annotation	Description	Example
@Digits	<p>@Digits annotation verifies the precision and scale of the field. The field must be a number within the specified range. The range is defined by the integer and the fraction elements. An integer and the fraction are required elements of an int type.</p>	<pre data-bbox="691 240 1176 304">@Digits (integer=7, fraction=2) private double monthlySale;</pre>
@DecimalMin	<p>@DecimalMin annotation verifies if the value in the field or property is a decimal value greater than or equal to the value defined in the DecimalMin. The value is a required element of the String type.</p>	<pre data-bbox="691 914 1029 977">@DecimalMin("8.5") private double minTax;</pre>
@DecimalMax	<p>@DecimalMax annotation verifies if the value in the field or property is a decimal value lesser than or equal to the value defined in the DecimalMax. The value is a required element of the String type.</p>	<pre data-bbox="691 1421 1029 1484">@DecimalMax("19.5") private double maxTax;</pre>

Annotation	Description	Example
@Future	@Future annotation verifies if the value in the field or property is a date in the future.	@Future private Date promotionDate;
@Past	@Past annotation verifies if the value in the field or property is a date in the past.	@Past private Date startDate;
@Pattern	@Pattern annotation verifies if the value in the field or property matches the regexp expression. The regular expression is a required element of the String type.	@Pattern (regexp="\\(\\d{3}\\)\\d{3}-\\d{4}") private String phoneNumber;
@AssertFalse	@AssertFalse annotation verifies if the value in the field or property is false.	@AssertFalse private boolean isAvailable;
@AssertTrue	@AssertTrue annotation verifies if the value in the field or property is true.	@AssertTrue private boolean isOrdered;

All bean validation annotations have optional attributes, such as the `message` attribute, which can be used to display a custom message if validation fails. Some annotations have required attributes. For instance, `DecimalMax` annotation has a `value` attribute of a `String` type to represent a maximum value. Some examples are as follows:

`@NotNull` with the `message` attribute can have a customized message that can be displayed instead of the default message if validation fails.

```
@NotNull(message="Address cannot be a null value")
private Address address;
```

## Automatic Versus Manual Invocation

### Automatic Invocation

Java EE 7 application server provides the Hibernate Validator package, which includes bean validation annotations as well as an automatic invocation of the validation constraints. When attaching an annotation to an entity field, Hibernate automatically validates that the data matches the annotation constraints placed on the fields. The following code snippet, for example, demonstrates using a `@Size(min=4)` constraint, applying it to the `personName` attribute of a `Person` class. When creating an instance of the entity, if the presented data does not conform to the validation constraint, in this case that the size of the String is at least four characters, then an error is returned. The application server and the validator framework automatically check the constraint before persisting an entity to the database.

```
...
//Using validation in constructor
public Person(@NotNull String personName){
 this.personName=personName;
}

//using validation in method parameter
public void setPersonName(@Size(min=4) String name){
 this.personName=personName;
}
...
```

### Manual Invocation

While many frameworks automatically validate entity fields based on these validation annotations, occasionally developers need to programmatically trigger bean validation. To programmatically validate an instance of an entity, use the `javax.validation.Validator` API. The `Validator` interface provides methods to validate an entire entity or a single property of an entity. The following code illustrates how to create a `ValidatorFactory` and `Validator` instance and use the `validator` to validate an object.

```
...
Person p = new Person();
p.setPersonName("RH");
ValidatorFactory validatorFactory = Validation.buildDefaultValidatorFactory();
Validator validator = validatorFactory.getValidator();
Set<ConstraintViolation<Person>> constraintViolations = validator.validate(p);
...
```

This code creates a set of `ContstraintViolations` that can be iterated over to view all of the violations that occur based on the entity's annotations. The following is an example of iterating over the `constraintViolations` set and logging each error:

```
for (ConstraintViolation<Person> cv : constraintViolations) {
 log.error(cv.getMessage());
}
```



## References

### Bean Validation API

<http://beanvalidation.org/>



## References

Further information is available in the *JBoss Application Server 7* guide for Red Hat JBoss EAP 7 at

<https://docs.jboss.org/author/display/AS71/Documentation/>

## ► Guided Exercise

# Validating Data

In this exercise, you will validate data in an entity class using bean validation.

### Outcomes

You should be able to validate a bean field using built-in constraints.

### Before You Begin

Open a terminal window on the `workstation` VM and run the following command to download the lab files required for this guided exercise.

```
[student@workstation ~]$ lab validate-data setup
```

## Steps

- 1. Import the `validate-data` project into the JBoss Developer Studio IDE (JBDS).
  - 1.1. Start JBDS by double-clicking the JBDS icon on the `workstation` VM desktop.
  - 1.2. In the Eclipse Launcher window, enter `/home/student/JB183/workspace` in the **Workspace** field, and then click **Launch**.
  - 1.3. In the JBDS menu, click **File > Import** to open the **Import** wizard.
  - 1.4. On the **Select** page, click **Maven > Existing Maven Projects**, and then click **Next**.
  - 1.5. In the **Maven projects** page, click **Browse** to open the **Select root folder** window. Navigate to the `/home/student/JB183/labs/` directory. Select the `validate-data` folder, and then click **OK**.
  - 1.6. On the **Maven projects** page, click **Finish**.
  - 1.7. Watch the JBDS status bar to monitor the progress of the import operation. It may take a few minutes to download all required dependencies.
- 2. Add validation constraints to the `Person` entity class to ensure that the `personName` is not blank.
  - 2.1. In the Project Explorer tab in the left pane of JBDS, select `validate-data > Java Resources > src/main/java > com.redhat.training.model` and expand the package.
  - 2.2. Double-click the `Person.java` file in the selected `com.redhat.training.model` package to open the class in the editor.
  - 2.3. Import the `javax.validation.constraints.NotNull` library and add the following `@NotNull` annotation to the `personName` attribute in the `Person` class:

```
//ToDo:add the validation imports
import javax.validation.constraints.NotNull;
.....
@Entity
public class Person implements Serializable{

 //TODO: Add validation constraints here

 @NotNull(message="Name cannot be blank")
 @Column(name="name")
 private String personName;

}
```

2.4. Save your changes by pressing **Ctrl+S**.

► **3.** Build and deploy the application.

- 3.1. Start EAP by selecting the **Servers** tab in the bottom pane of JBDS. Right-click the server **Red Hat JBoss EAP 7.0 [Stopped]** and click on the green start button to start the server.
- 3.2. Build the validate-data application using the following commands in the terminal window:

```
[student@workstation ~]$ cd /home/student/JB183/labs/validate-data
[student@workstation validate-data]$ mvn clean package
```

- 3.3. Deploy the validate-data application using the following command in the terminal window:

```
[student@workstation validate-data]$ mvn wildfly:deploy
```

► **4.** Test the application for blank field validation.

- 4.1. Use a web browser on the **workstation** VM to navigate to **http://localhost:8080/validate-data/** to access the validate-data application.
- 4.2. Do not enter any data in the **Enter your name** field and click **Submit**.
- 4.3. Verify that the field is blank and see the response with an error message **Name cannot be blank**.

# Validate Data web app

Enter your name:

- Name cannot be blank

Figure 4.9: The validate-data application's blank name response

- 4.4. Enter **James** in the Enter your name field and click Submit.
- 4.5. Verify that the server processes the input and responds with a greeting using the name you entered.

# Validate Data web app

Enter your name:

- Hello JAMES!. Time on the server is: Oct 26 2017 09:34:32 PM

Figure 4.10: The validate-data application response

- 5. Add a validation constraint to the Person entity class to test that the personName is not less than two characters.
- 5.1. Import the `javax.validation.constraints.Size` library. Add the `@Size(min=2, message="Name cannot be less than 2 characters")` annotation to personName attribute in the Person class.

```
//ToDo:add the validation imports
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
...
@Entity
public class Person implements Serializable{
```

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;

//TODO: Add validation constraints here

@NotNull(message="Name cannot be blank")
@Size(
min=2,
message="Name cannot be less than 2 characters"
)
@Column(name="name")
private String personName;
....
}
```

5.2. Save your changes by pressing **Ctrl+S**.

► **6.** Rebuild and redeploy the application.

6.1. Rebuild the validate-data application using the following commands in the terminal window:

```
[student@workstation validate-data]$ mvn clean package
```

6.2. Redeploy the validate-data application using the following command in the terminal window:

```
[student@workstation validate-data]$ mvn wildfly:deploy
```

► **7.** Test the application for validation of the size of the name field.

- 7.1. Use a web browser in the workstation VM to navigate to `http://localhost:8080/validate-data/` to access the validate-data application.
- 7.2. Enter one character in the **Enter your name** field and click **Submit**.
- 7.3. Verify that the message **Name cannot be less than 2 characters** displayed as the response.

# Validate Data web app

Enter your name:

- Name cannot be less than 2 characters

Figure 4.11: The validate-data application's validation of length

- 7.4. Enter **A1** in the Enter your name field and click Submit.

Verify that the server processes the input and responds with a greeting using the name you entered and the current time on the server.

► 8. Undeploy the application and stop EAP.

- 8.1. In the terminal window where you ran the Maven command to deploy the application, run the following command to undeploy the application:

```
[student@workstation validate-data]$ mvn wildfly:undeploy
```

- 8.2. Right-click on the validate-data project in the Project Explorer, and select Close Project to close this project.

- 8.3. Right-click Red Hat JBoss EAP 7.0 in the Servers tab and then click Stop to stop the EAP instance.

This concludes the guided exercise.

# Creating Queries

---

## Objective

After completing this section, students should be able to create queries using Java Persistence Query Language.

## Creating Queries

Java Persistence Query Language (JPQL) is a platform-independent query language defined as a part of the JPA specification to perform queries on entities in an object-oriented manner. JPQL is similar to SQL in syntax, but JPQL queries are expressed in terms of Java entities rather than database tables and columns. JPA providers, such as Hibernate, transform JPQL queries to SQL. JPQL supports the SELECT, UPDATE, and DELETE statements. To understand how to create different types of queries with JPQL, consider an example of an employee entity class:

```
@Entity
public class Employee implements Serializable{
 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;
 private String empName;
 private double salary;
 ...
}
```

An Employee table is created in a database for an Employee entity. The sample data of all employees from an Employee table is shown below:

```
MariaDB [todo]> select * from Employee;
+----+-----+-----+
| id | empName | salary |
+----+-----+-----+
| 1 | Tim | 120000 |
| 2 | Joe | 100000 |
| 3 | Tom | 125000 |
| 4 | Mia | 135000 |
| 5 | Matt | 145000 |
| 6 | Kim | 115000 |
| 7 | Nita | 117000 |
+----+-----+-----+
7 rows in set (0.00 sec)
```

The JPQL query to retrieve records of all employees from the database is as follows:

```
SELECT e FROM Employee e;
```

The EntityManager API supports methods for creating both static and dynamic queries. The `createNamedQuery` method is used to create static queries, whereas the `createQuery` method is used to create dynamic queries.

Dynamic queries are created at runtime by an application using the following process:

1. Create a string containing a JPQL query.
2. Pass the string to the entity manager's `createQuery` method and store the returned `Query` object.
3. Use the query's `getResultSet()` method to execute the query and return the selected rows from the database.

```
...
String simpleQuery="SELECT e from Employee e";
Query query=entityManager.createQuery(simpleQuery);
List<Employee> persons = query.getResultSet();
...
```

The query retrieves all employees in a database table.

```
1, Tim, 120000.0
2, Joe, 100000.0
3, Tom, 125000.0
4, Mia, 135000.0
5, Matt, 145000.0
6, Kim, 115000.0
7, Nita, 117000.0
```

Database functions such as LOWER, UPPER, LENGTH, as well as arithmetic functions, can also be applied to JPQL queries:

```
...
public List<String> getAllNames(){
 Query query=entityManager.createQuery("SELECT UPPER(e.empName) from Employee e");
 List<String> names = query.getResultSet();
 return names;
}
```

This code displays all Employee names in uppercase:

```
Output:
TIM
JOE
TOM
MIA
MATT
KIM
NITA
```

To provide type safety, JPA also supports the `TypedQuery<?>` class, which allows static typing of queries to avoid any issues with casting results. To create a `TypedQuery`, pass the class that should match the type of the query results into the `createQuery` method.

The following example shows a type-safe TypedQuery:

```
TypedQuery<Employee> query=entityManager.createQuery("SELECT e from Employee e
where e.salary >?1 or e.empName=?2", Employee.class);
```

JPQL also supports arithmetic functions in queries. The following is an example to get the sum and an average of the salaries of all employees:

```
...
public String[] getSumAndAvgSalary(){
 Query query=entityManager.createQuery("SELECT sum(e.salary), round(avg(e.salary),2)
from Employee e");
 Object[] sal =(Object[])query.getSingleResult();
 String[] s= {"Sum of all salaries :" +sal[0], "avg of all salaries :
"+sal[1]};
 return s;
}
```

This code results in the following output based on the given data set:

**Output:**

```
Sum of all salaries: 857000.0
Average of all salaries: 122428.57
```

Note that in the previous example, the query contains two fields: one for the sum of the salaries and the other for the average salary. When multiple fields are returned as a result of the query, the `getSingleResult` method returns an `Object` array.

The results of the queries are filtered using the `WHERE` clause in the queries. The `WHERE` clause is used to define the conditions on the data that the query returns. To create a conditional expressions for the query, various operators can be used. Operators available in SQL are also available in JPQL. The operands in the condition depends on the expression. Commonly used operators are:

- `<`, `=`, `>`, `<=`, `>=`, `<>` are used to compare the arithmetic values.
- `IN` and `NOT IN` are used for all types. `IN` operator is used to determine whether the data in a field is one of the values provided in a list of values.
- `LIKE` and `NOT LIKE` are used for string values. It is used to determine whether data in a field matches a sequence of characters provided in the string.
- `BETWEEN` and `NOT BETWEEN` are used for arithmetic, date, time, and string values. It is used to determine whether data in a field lies in a certain range of values.
- `MEMBER OF`, `NOT MEMBER OF`, `IS EMPTY`, and `IS NOT EMPTY` are used for Collection types.

Special characters like an `_` (underscore) for any single character and a `%` character for any character sequence can be used to build the string expressions. A simple example of a query with a `WHERE` clause to print all employees whose salary is greater than `120000` is as follows:

```
Query query=entityManager.createQuery("SELECT e from Employee e where e.salary >120000");
List<Employee> persons = query.getResultList();
```

This code produces the following output based on the given data set:

```
Output:
3, Tom, 125000.0
4, Mia, 135000.0
5, Matt, 145000.0
```

## Named Parameters in Queries

Queries with the WHERE clause can be created with *named parameters* in JPQL. A named parameter is a query parameter serving as a placeholder for real values. A query can be reused and executed with a different set of data provided at runtime for a named parameter. A named parameter is prefixed with a : character.

A named parameter is bound to the arguments by using the `setParameter()` method of `javax.persistence.Query` API. The first parameter in the `setParameter()` method is the name of a named parameter. The second parameter is the value for the named parameter. An example of JPQL query with a named parameter is:

```
...
public List<Employee> getEmployeesWithGreaterSalary(double salary) {
 Query query=entityManager.createQuery("SELECT e from Employee e where e.salary
 >:sal");
 query.setParameter("sal", salary);
 List<Employee> persons = query.getResultList();

 return persons;
}
```

When a user inputs the value 115000 for the salary, the code returns the following output:

```
Output:
1, Tim, 120000.0
3, Tom, 125000.0
4, Mia, 135000.0
5, Matt, 145000.0
7, Nita, 117000.0
```

## Positional Parameters in Queries

The *positional parameters* are query parameters in the form of an index or the ordinal position of a parameter in the query. These can be passed to queries as an alternative to named parameters, depending on the developer's preference for readability. Positional parameters are prefixed with ? followed by the numeric position of the parameter in the query.

The `setParameter()` method is used to bind a positional parameter to a query. The values for a positional parameter is provided at runtime. The first parameter in the `setParameter()` method is the position of a parameter in the query and the second parameter is a variable containing the

value for the parameter. An example of a JPQL query with one positional parameter where the value of the salary for the query is provided in the first positional parameter as ?1 is as follows:

```
...
public List<Employee> getAllPersonsWithPositionParam(double salary) {
 Query query=entityManager.createQuery("SELECT e from Employee e where e.salary >?
1");
 query.setParameter(1, salary);
 return query.getResultList();
}
```

When the user inputs the value 130000 for the salary, the code produces the following output based on the given data set:

Output:

```
4, Mia, 135000.0
5, Matt, 145000.0
```

The following is an example of a type-safe query with two positional parameters, where the first positional parameter ?1 refers to the salary and the second positional parameter ?2 refers to the name in the query:

```
...
public List<Employee> getAllPersons(double salary, String name) {
 TypedQuery<Employee> query=entityManager.createQuery("SELECT e from Employee e
where e.salary >?1 or e.empName=?2", Employee.class);
 query.setParameter(1, salary);
 query.setParameter(2, name);
 return query.getResultList();
}
```

When the user inputs the values 130000 for the salary and Tim for the name, the code produces the following output:

Output:

```
1, Tim, 120000.0
4, Mia, 135000.0
5, Matt, 145000.0
```

## Named Queries

A *named query* is a predefined query attached to an entity. Named queries are parsed at startup so that errors can be detected quickly. Another advantage is that the code and the queries are separate. Named queries are defined by using the `javax.persistence.NamedQuery` annotation. The `@NamedQuery` annotation can be applied at the entity's class level. The `@NamedQuery` annotation has four elements: `name`, `query`, `hints`, and `lockMode`.

- A name is a required element of the `NamedQuery` annotation. It defines the name that is used by the `EntityManager` methods to refer to a query. The name of the named query must be unique, as the scope of the named query is the persistence unit.

## Chapter 4 | Managing Persistence

- The **query** is a required element of the `NamedQuery` annotation and represents the JPQL query string.
- The **hints** element is an optional element of the `NamedQuery` annotation. It represents query hints and properties. These hints can be vendor-specific.
- The **lockMode** element is an optional element of the `NamedQuery` annotation. It represents the lock mode type to use in query execution. When lock mode type is defined as anything other than `NONE`, the query must be executed in a transaction.

The named query to view all employees with a salary greater than the value input by the user is defined in the `Employee` entity class:

```
@Entity
@NamedQuery(
 name="getAllEmployees",
 query="select e from Employee e where e.salary > :sal")
public class Employee implements Serializable{

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;
 private String empName;
 private double salary;
 ...
}
```

To execute the named query, use the `createNamedQuery()` method of the `EntityManager` to create the query and set the parameters:

```
...
public List<Employee> getPersonsWithNamedQuery(double salary) {
 Query query=entityManager.createNamedQuery("getAllEmployees")
 query.setParameter("sal", salary);
 return query.getResultList();
}
```

To define more than one named queries for an entity, the `@NamedQueries` annotation is used. It acts as a wrapper for multiple queries. The `@NamedQueries` annotation is applied at the entity's class level.

```
@Entity
@NamedQueries({
 @NamedQuery(name="getAllEmployees",
 query="select e from Employee e where e.salary > :sal"),
 @NamedQuery(name="getEmployeesWithSalaryOrName",
 query="select e from Employee e where e.salary > :sal or e.empName=:name")
})
public class Employee implements Serializable{

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;
```

```
private String empName;
private double salary;
....
```

The `createNamedQuery()` method creates the query. The parameters are set as follows:

```
public List<Employee> getAllPersonsWithNamedQueries(double salary, String ename) {

 Query query=entityManager.createNamedQuery("getEmployeesWithSalaryOrName");
 query.setParameter("sal", salary);
 query.setParameter("name", ename);
 List<Employee> persons = query.getResultList();

 return persons;
}
```

## ► Guided Exercise

# Creating Queries

In this exercise, you will create queries with named parameters and positional parameters as well as a named query to retrieve data from the database.

## Outcomes

You should be able to create named queries and create queries with both named parameters and positional parameters.

## Before You Begin

Open a terminal window on the **workstation** VM and run the following command to download the lab files required for this guided exercise.

```
[student@workstation ~]$ lab create-queries setup
```

## Steps

- 1. Import the `create-queries` project into the JBoss Developer Studio IDE (JBDS).
  - 1.1. Start JBDS by double-clicking the JBDS icon on the **workstation** VM desktop.
  - 1.2. In the Eclipse Launcher window, enter **/home/student/JB183/workspace** in the **Workspace** field, and then click **Launch**.
  - 1.3. In the JBDS menu, click **File > Import** to open the **Import** wizard.
  - 1.4. On the **Select** page, click **Maven > Existing Maven Projects**, and then click **Next**.
  - 1.5. In the **Maven projects** page, click **Browse** to open the **Select root folder** window. Navigate to the `/home/student/JB183/labs/` directory. Select the `create-queries` folder, and then click **OK**.
  - 1.6. On the **Maven projects** page, click **Finish**.
  - 1.7. Watch the JBDS status bar to monitor the progress of the import operation. It may take a few minutes to download all required dependencies.
- 2. Add a method `getAllPersons()` in the `PersonService` class to view all persons in the `Person` database table.
  - 2.1. In the **Project Explorer** tab in the left pane of JBDS, select `create-queries > Java Resources > src/main/java > com.redhat.training.services`, and expand the package.
  - 2.2. Double-click the `PersonService.java` file in the `com.redhat.training.services` package to open the `PersonService` class in the editor pane.
  - 2.3. To view the data of all persons in a database table, add a new method `getAllPersons` with a simple query:

```
// Get all Person objects in the Database
public List<Person> getAllPersons() {
 TypedQuery<Person> query = entityManager.createQuery("SELECT p FROM Person p",
 Person.class);
 return query.getResultList();
}
```

- 2.4. Save your changes by pressing **Ctrl+S**.
- 3. In the `Hello.java` bean in the `com.redhat.training.ui` package, uncomment the `getPersons()` method to add the functionality to view the names of all `Person` objects from the database.
- 3.1. In the **Project Explorer** tab in the left pane of JBDS, select **create-queries > Java Resources > src/main/java > com.redhat.training.ui**, and expand the package.
  - 3.2. Double-click the `Hello.java` file in the `com.redhat.training.ui` package to view the class in the editor pane.
  - 3.3. Remove the comments on the `getPersons()` method.

```
//View all persons in the database table
public List<Person> getPersons() {
 return personService.getAllPersons();
}
```

- 3.4. Save your changes by pressing **Ctrl+S**.
- 4. Build and deploy the application.
- 4.1. Start EAP by selecting the **Servers** tab in the bottom pane of JBDS. Right-click the server **Red Hat JBoss EAP 7.0 [Stopped]** and click on the green start button to start the server.
  - 4.2. Build the `create-queries` application using the following commands in the terminal window:

```
[student@workstation ~]$ cd /home/student/JB183/labs/create-queries
[student@workstation create-queries]$ mvn clean package
```

- 4.3. Deploy the `create-queries` application using the following command in the terminal window:
- ```
[student@workstation create-queries]$ mvn wildfly:deploy
```
- 5. Test the application to view all persons.
- 5.1. Use a web browser on the `workstation` VM to navigate to `http://localhost:8080/create-queries/` to access the `create-queries` application.
 - 5.2. Click **View All Users** to view a list of the `Person` objects in the database.

- 1 John
- 2 Steve
- 3 Mark
- 4 Nick
- 5 Nita
- 6 John
- 7 Julia
- 8 Joe
- 9 Nita
- 10 Branden

Figure 4.12: The list of names in the database

- 6. Create a query with named parameters to fetch all Person objects from the database table that match a specific name.

- 6.1. Click the **PersonService.java** file to edit the **PersonService** class in the editor pane.
- 6.2. Add a new method **getPersonsWithName(String name)** with a named parameter:

```
//Get persons whose name matches the name given in the query
public List<Person> getPersonsWithName(String name) {
    TypedQuery<Person> query = entityManager.createQuery("SELECT p FROM Person p
where p.name =:pname", Person.class);
    query.setParameter("pname", name);
    return query.getResultList();
}
```

- 6.3. Save your changes by pressing **Ctrl+S**.

- 7. In the **Hello.java** class in the **com.redhat.training.ui** package, uncomment the **search()** method to view the names matching the user's search.

- 7.1. Double-click the **Hello.java** file in the **com.redhat.training.ui** package to view the class in the editor pane.
- 7.2. Remove the comments on the **search()** method.

```
//view all persons whose name matches the name given in the query
public void search() {
    results = personService.getPersonsWithName(name);
}
```

- 7.3. Save your changes by pressing **Ctrl+S**.

- 8. Deploy the **create-queries** application using the following command in the terminal window:

```
[student@workstation create-queries]$ mvn wildfly:deploy
```

- 9. Test the search feature of the application to verify that the query returns **Person** objects by name.

- 9.1. Use a web browser on the **workstation** VM to navigate to `http://localhost:8080/create-queries/` to access the `create-queries` application.
 - 9.2. Click **Search Users** to navigate to the search page.
 - 9.3. In the **Name** field, enter a name that already exists in the database, such as **John**, and click **Submit**. Notice that the query returns both entries for **John**.
- ▶ 10. Modify the existing query to use positional parameters to fetch **Person** objects from the database table.
- 10.1. Click the **PersonService.java** file to edit the **PersonService** class in the editor pane.
 - 10.2. Update the `getPersonsWithName()` method in the **PersonService** class to use positional parameters:

```
//Get persons whose name matches the name given in the query
public List<Person> getPersonsWithName(String name) {
    TypedQuery<Person> query = entityManager.createQuery("SELECT p FROM Person p
    WHERE p.name =?1", Person.class);
    query.setParameter(1, name);
    return query.getResultList();
}
```

- 10.3. Save your changes by pressing **Ctrl+S**.
- ▶ 11. Deploy the `create-queries` application using the following command in the terminal window:
- ```
[student@workstation create-queries]$ mvn wildfly:deploy
```
- ▶ 12. Test the search feature that uses a positional parameter query.
- 12.1. Use a web browser on the **workstation** VM to navigate to `http://localhost:8080/create-queries/` to access the `create-queries` application.
  - 12.2. Click **Search Users** to navigate to the search page. In the **Name** field, enter a name that already exists in the database, such as **Nita**, and click **Submit**. Verify that the server returns the correct results.
- ▶ 13. Create a named query to fetch **Person** objects from a database table.
- 13.1. In the **Project Explorer** tab in the left pane of JBDS, select `create-queries > Java Resources > src/main/java > com.redhat.training.model`, and expand the package.
  - 13.2. Double-click the **Person.java** file in the `com.redhat.training.model` package to open the **Person** class in the editor pane.
  - 13.3. Click the **Person.java** file to edit **Person** class in the editor pane.
  - 13.4. Add the following `NamedQuery` annotation to create a named query to fetch the **Person** objects by name:

```
@Entity
//add named query here
@NamedQuery(
```

```
name="getAllPersonsWithName",
query="select p from Person p where p.name = :pname")
public class Person{

 @Id
 private Long id;

 private String name;
 ...
}
```

13.5. Save your changes by pressing **Ctrl+S**.

13.6. In the `PersonService.java` class, update the `getPersonsWithName` method to use the `getAllPersonsWithName` named query:

```
//Get persons whose name matches the name given in the query
public List<Person> getPersonsWithName(String name) {
 TypedQuery
 query=entityManager.createNamedQuery("getAllPersonsWithName",Person.class);
 query.setParameter("pname", name);
 return query.getResultList();
}
```

13.7. Save your changes by pressing **Ctrl+S**.

► 14. Deploy the `create-queries` application using the following command in the terminal window:

```
[student@workstation create-queries]$ mvn wildfly:deploy
```

► 15. Test the search feature of the application using a named query.

- 15.1. Use a web browser in the `workstation` VM to navigate to `http://localhost:8080/create-queries/` to access the `create-queries` application.
- 15.2. Click **Search Users** to navigate to the search page. In the **Name** field, enter a name that already exists in the database, such as `John`, and click **Submit**. Verify that the server returns the expected results.

► 16. Undeploy the application and stop EAP.

- 16.1. In the terminal window where you ran the Maven command to deploy the application, run the following command to undeploy the application:

```
[student@workstation create-queries]$ mvn wildfly:undeploy
```

- 16.2. Right-click on the `create-queries` project in the **Project Explorer**, and select **Close Project** to close this project.
- 16.3. Right-click **Red Hat JBoss EAP 7.0** in the **Servers** tab and then click **Stop** to stop the EAP instance.

This concludes the guided exercise.

## ▶ Lab

# Managing Persistence

In this lab, you will implement persistence in the To Do List Application using JPA.

## Outcome

You should be able to inject an entity manager using the default persistence context, and then implement persistence functionality using the entity manager API.

## Before You Begin

Open a terminal window on the `workstation` VM and run the following command to download the files required for this lab.

```
[student@workstation ~]$ lab persistence-lab setup
```

1. Open JBDS and import the `persistence-lab` project located in the `/home/student/JB183/labs/persistence-lab` directory.
2. Review the JNDI name assigned to the lab defined for the MySQLDS in the JBoss configuration file.
3. Review the `persistence.xml` file that defines the persistence context for the `persistence-lab` application.
4. Update the `Item` model class to be an entity that is managed by JPA. Mark the `id` field as the unique identifier field, as well as a generated value.
5. Inject an entity manager that uses the default persistence context into the `ItemRepository` class.

Then update the class to use the entity manager to implement the `findById` method functionality so that it no longer returns `null`.

Additionally, update the `findAllItems` method to create a query using JPQL that selects the items sorted by whether or not they are completed (using the `done` field).

6. Update the `ItemService` EJB class to inject an entity manager associated with the default persistence context.

Then implement the `register(Item item)`, `remove(Long id)` and `update(Item item)` methods to use the appropriate entity manager methods.

**Hint: For the remove method, use the `ItemRepository.findById()` method to retrieve the Item first, then you can remove the instance using the entity manager.**

7. Start JBoss EAP from within JBDS.
8. Build and deploy the application to JBoss EAP using Maven.
9. Test the application in a browser.
  - 9.1. Open Firefox on the `workstation` VM, and navigate to `http://localhost:8080/persistence-lab` to access the application.

- 9.2. Add at least two new to-do items using the To Do List application interface. If persistence is working properly, the new items are added to the list.
  - 9.3. After the new items are successfully added, update the status of those items to be completed. If the query is functioning properly, items marked as done are moved to the bottom of the list.
- 10.** Open a new terminal window and run the following command to grade the lab:

```
[student@workstation ~]$ lab persistence-lab grade
```

The grading script should report SUCCESS. If there is a failure, check the errors and fix them until you see a SUCCESS message.

- 11.** Clean up.

11.1. Undeploy the application from JBoss EAP using Maven with the following commands:

```
[student@workstation ~]$ cd /home/student/JB183/labs/persistence-lab
[student@workstation persistence-lab]$ mvn wildfly:undeploy
```

- 11.2. Right-click on the **persistence-lab** project in the **Project Explorer**, and select **Close Project** to close this project.
- 11.3. Right-click on the Red Hat JBoss EAP 7.0 server in the **JBDS Servers** tab and click **Stop**.

This concludes the lab.

## ► Solution

# Managing Persistence

In this lab, you will implement persistence in the To Do List Application using JPA.

## Outcome

You should be able to inject an entity manager using the default persistence context, and then implement persistence functionality using the entity manager API.

## Before You Begin

Open a terminal window on the **workstation** VM and run the following command to download the files required for this lab.

```
[student@workstation ~]$ lab persistence-lab setup
```

1. Open JBDS and import the **persistence-lab** project located in the `/home/student/JB183/labs/persistence-lab` directory.
  - 1.1. Double-click the JBoss Developer Studio icon on the workstation desktop to open JBDS. Leave the default workspace (`/home/student/JB183/workspace`) and click **OK**.
  - 1.2. In the JBDS menu, click **File > Import** to open the **Import** wizard.
  - 1.3. On the **Select** page, click **Maven > Existing Maven Projects**, and then click **Next**.
  - 1.4. In the **Maven projects** page, click **Browse** to open the **Select root folder** window. Navigate to the `/home/student/JB183/labs/` directory. Select the **persistence-lab** folder, and then click **OK**.
  - 1.5. On the **Maven projects** page, click **Finish**.
  - 1.6. Watch the JBDS status bar to monitor the progress of the import operation. It may take a few minutes to download all required dependencies.
2. Review the JNDI name assigned to the lab defined for the MySQLDS in the JBoss configuration file.
  - 2.1. Using your preferred text editor, open the EAP configuration file at `/opt/eap/standalone/configuration/standalone-full.xml`:

```
[student@workstation ~]$ less /opt/eap/standalone/configuration/\
standalone-full.xml
```

- 2.2. Navigate to the `urn:jboss:domain:datasources:4.0` subsystem:

```
<subsystem xmlns="urn:jboss:domain:datasources:4.0">
 <datasources>
 <datasource jndi-name="java:jboss/datasources/MySQLDS" pool-name="MySQLDS">
```

```
<connection-url>jdbc:mysql://172.25.250.10:3306/todo</connection-url>
<driver>mysql</driver>
<security>
 <user-name>todo</user-name>
 <password>todo</password>
</security>
</datasource>
</datasources>
</subsystem>
```

Note that the MySQLDS has a JNDI entry of `java:jboss/datasources/MySQLDS`.

- 2.3. Close the text editor and **do not** save any changes to the `standalone-full.xml` file.



### Warning

Problems can occur in the lab if unexpected changes are introduced in the configuration file.

3. Review the `persistence.xml` file that defines the persistence context for the persistence-lab application.

Open the `persistence.xml` file by expanding the `persistence-lab` item in the **Project Explorer** tab in the left pane of JBDS, then click on `persistence-lab > Java Resources > src/main/resources > META-INF` to expand it. Double-click 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="Items" transaction-type="JTA">
 <jta-data-source>java:jboss/datasources/MySQLDS</jta-data-source>
 <properties>
 <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect" />
 <property name="hibernate.hbm2ddl.auto" value="update" />
 <property name="hibernate.show_sql" value="true" />
 <property name="hibernate.format_sql" value="true" />
 </properties>
 </persistence-unit>
</persistence>
```

4. Update the `Item` model class to be an entity that is managed by JPA. Mark the `id` field as the unique identifier field, as well as a generated value.

- 4.1. Open the `Item` class by expanding the `persistence-lab` item in the **Project Explorer** tab in the left pane of JBDS, then click on `persistence-lab > Java Resources > src/main/java > com.redhat.training.todo.model` to expand it. Double-click the `Item` file.
- 4.2. Add the `@Entity` annotation at the class level to mark the `Item` class as an entity managed by JPA that maps to a database table named 'Item':

```
//TODO mark the Item class as an entity to be managed by JPA, mapped to a table
// named 'Item'
@Entity
public class Item {
```

- 4.3. Add the `@Id` and `@GeneratedValue` annotations on the `id` field to mark it as the unique identifier for JPA.

```
@Entity
public class Item {

 //TODO mark this field as the unique identifier for JPA as well as a generated
 // value
 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;
```

- 4.4. Save your changes using **Ctrl+S**.

5. Inject an entity manager that uses the default persistence context into the `ItemRepository` class.

Then update the class to use the entity manager to implement the `findById` method functionality so that it no longer returns `null`.

Additionally, update the `findAllItems` method to create a query using JPQL that selects the items sorted by whether or not they are completed (using the `done` field).

- 5.1. Open the `ItemRepository` class by expanding the `persistence-lab` item in the **Project Explorer** tab in the left pane of JBDS, then click on `persistence-lab > Java Resources > src/main/java > com.redhat.training.todo.data` to expand it. Double-click the `ItemRepository` file.

- 5.2. Inject an entity manager that uses the default persistence context using the `@PersistenceContext` annotation:

```
@ApplicationScoped
public class ItemRepository {

 //TODO Inject EntityManager using the default persistence context
 @PersistenceContext
 private EntityManager em;
```

- 5.3. Implement the `findById` method using the entity manager's `find` method:

```
public Item findById(Long id) {
 //TODO use the entity manager to implement the findById method
 return em.find(Item.class, id);
}
```

- 5.4. Update the `findAllItems` method to create a query using JPQL that uses the `ORDER BY` keyword to sort the results:

```

public List<Item> findAllItems() {
 //TODO Create a query to select all the items in order by whether or not they are
 done
 TypedQuery<Item> query = em.createQuery("SELECT i FROM Item i ORDER BY i.done",
 Item.class);

 return query.getResultList();
}

```

- 5.5. Save your changes using **Ctrl+S**.
6. Update the `ItemService` EJB class to inject an entity manager associated with the default persistence context.

Then implement the `register(Item item)`, `remove(Long id)` and `update(Item item)` methods to use the appropriate entity manager methods.

**Hint: For the remove method, use the `ItemRepository.findById()` method to retrieve the Item first, then you can remove the instance using the entity manager.**

- 6.1. Open the `ItemService` class by expanding the `persistence-lab` item in the Project Explorer tab in the left pane of JBDS, then click on `persistence-lab > Java Resources > src/main/java > com.redhat.training.todo.service` to expand it. Double-click the `ItemService` file.
- 6.2. Inject an entity manager that uses the default persistence context using the `@PersistenceContext` annotation:

```

import javax.persistence.PersistenceContext;

...
@Stateless
public class ItemService {

 @Inject
 private Logger log;

 //TODO Inject an entity manager using the default persistence context
 @PersistenceContext
 private EntityManager em;
}

```

- 6.3. Implement the `register` method using the `persist` method on the entity manager:

```

public void register(Item item) throws Exception {
 log.info("Adding new task: " + item.getDescription());
 //TODO persist item with the entity manager
 em.persist(item);
}

```

- 6.4. Implement the `remove` method using the `findById` method, and then pass the `Item` that was found to the entity manager's `remove` method.

```
public void remove(Long id) {
 //TODO lookup the item by its ID then remove it using the entity manager
 em.remove(repository.findById(id));
}
```

- 6.5. Implement the update method using the merge method on the entity manager to update existing items in the database:

```
public void update(Item item) {
 //TODO update the item in the database using the entity manager
 em.merge(item);
}
```

- 6.6. Save your changes using Ctrl+S.

7. Start JBoss EAP from within JBDS.

Select the **Servers** tab in JBDS. Right-click on the server entry **Red Hat JBoss EAP 7.0 [Stopped]** and click the green Start option to start the server. Watch the **Console** tab of JBDS until the server starts and you see the message:

```
INFO [org.jboss.as] (Controller Boot Thread) WFLYSRV0025: JBoss EAP 7.0.2.GA
(WildFly Core 2.1.8.Final-redhat-1) started
```

8. Build and deploy the application to JBoss EAP using Maven.

Run the following commands to build and deploy the application:

```
[student@workstation ~]$ cd /home/student/JB183/labs/persistence-lab
[student@workstation persistence-lab]$ mvn clean wildfly:deploy
```

A successful build displays the following output:

```
[student@workstation persistence-lab]$ mvn clean wildfly:deploy
...
[INFO] [INFO] <<< wildfly-maven-plugin:1.0.2.Final:deploy (default-cli) < package
@ persistence-lab <<<
...
[INFO] --- maven-war-plugin:2.1.1:war (default-war) @ persistence-lab ---
...
[INFO] --- wildfly-maven-plugin:1.0.2.Final:deploy (default-cli) @ persistence-lab
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
```

9. Test the application in a browser.

- 9.1. Open Firefox on the workstation VM, and navigate to <http://localhost:8080/persistence-lab> to access the application.
- 9.2. Add at least two new to-do items using the To Do List application interface. If persistence is working properly, the new items are added to the list.

- 9.3. After the new items are successfully added, update the status of those items to be completed. If the query is functioning properly, items marked as **done** are moved to the bottom of the list.
- 10.** Open a new terminal window and run the following command to grade the lab:

```
[student@workstation ~]$ lab persistence-lab grade
```

The grading script should report **SUCCESS**. If there is a failure, check the errors and fix them until you see a **SUCCESS** message.

- 11.** Clean up.

11.1. Undeploy the application from JBoss EAP using Maven with the following commands:

```
[student@workstation ~]$ cd /home/student/JB183/labs/persistence-lab
[student@workstation persistence-lab]$ mvn wildfly:undeploy
```

11.2. Right-click on the **persistence-lab** project in the **Project Explorer**, and select **Close Project** to close this project.

11.3. Right-click on the Red Hat JBoss EAP 7.0 server in the JBDS **Servers** tab and click **Stop**.

This concludes the lab.

# Summary

---

In this chapter, you learned:

- Java Persistence API (JPA) specification supports object-relational mapping and has three key concepts: entities, persistence unit, and persistence context.
- A simple Java class is converted to an entity class by using an `@Entity` annotation.
- An entity class must specify a primary key field by using `@Id` annotation.
- The `EntityManager` performs the actual CRUD (Create, Read, Update, and Delete) operations using `persist()`, `find()`, `update()`, and `delete()` methods. An `EntityManager` object can be injected to an EJB using `@Inject` annotation.
- A persistence unit contains information about a data source, transactions type and the persistence unit name. It is configured in a `persistence.xml` file.
- Bean validation API configures validation constraints with annotations. Built-in constraints are available for validation in `javax.validation.constraints` package.
- JPQL is a query language that supports dynamic and static queries. Queries can be created with named parameters and positional parameters.
- Named queries are defined at the class level.



## Chapter 5

# Managing Entity Relationships

### Overview

**Goal** Define and manage JPA entity relationships.

**Objectives**

- Configure one-to-one and one-to-many entity relationships.
- Describe many-to-many entity relationships.

**Sections**

- Configuring Entity Relationships (and Guided Exercise)
- Describing Many-to-Many Entity Relationships (and Quiz)

**Lab** Managing Entity Relationships

# Configuring Entity Relationships

## Objective

After completing this section, students should be able to configure one-to-one and one-to-many entity relationships.

## Understanding Relationships Between Entities

When building enterprise applications, developers use relational databases to store business data created and updated using the application. Application data typically spans multiple database tables, so it is common for data in one table to need to reference data in another.

Consider the following example: a database storing customer order data must have three tables, one for customers, one for items, and one for the customer's orders. The order table needs to reference both a customer record, as well as an item record.

To represent these relationships, databases use what is called a *foreign key*. A foreign key is a column where the value of the data in the column is a reference to the ID or primary key of a row in another table. When a client loads the order record, the data in the foreign key column is used to retrieve the customer information as well as the item information. By referencing the data directly in the customer and item tables by their primary keys, there is no need to duplicate that data in the order table.

When representing database tables in Java EE, developers use entity beans, one for each table. To create a relationship between two entities, class-level variables are used to represent an instance of one entity as an attribute of another entity.

The following example shows sample Java entity beans for the customer order data described previously:

### Customer Entity:

```
@Entity
public class Customer {

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;

 private String name;

 private String email;
 ...}
```

### Item Entity:

```
@Entity
public class Item {
 @Id
```

```

@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;

private String name;

private String description;

private Double price;
...

```

**Order Entity:**

```

@Entity
public class Item {

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;

 private Customer customer;

 private Item item;
...

```

Notice that the `Customer` and `Item` classes are used as attributes of the `Order` class. This, from a Java class perspective, represents the relationship between those entities; each `Order` has a `Customer` and an `Item`.

After the entity relationships are properly represented using class-level variables on the entity beans, developers use annotations to control JPA and properly map those entities and their relationships when retrieving data from the database. The following table summarizes these annotations, all of which are covered in depth with examples later in the chapter:

**Standard JPA Relationship Annotations**

Annotation	Description
<code>@OneToOne</code>	Defines an entity relationship as a single value, where one row in table X is related to a single row in table Y, and vice versa.
<code>@OneToMany</code>	Defines an entity relationship as multi-valued, where one row in table X can be related to one or many rows in table Y.
<code>@ManyToOne</code>	Defines an entity relationship as multi-valued, where many rows in table X can be related to a single row in table Y.
<code>@ManyToMany</code>	Defines an entity relationship as multi-valued, where many rows in table X can be related to a one or many rows in table Y.
<code>@JoinColumn</code>	Defines the column that JPA uses as the foreign key.

## Using a One-to-one Entity Relationship

Use the `@OneToOne` annotation when two entities relate to each other such that an instance of one entity only relates to a single instance of the other entity. For example, if an application stores user information, but places sensitive information, such as a social security number, in a separate table, then two entities such as `User` and `UserSSN` are related using a `@OneToOne` annotation. Using the `@OneToOne` relationship, each user has a single SSN, and each SSN has a single user.

The following example demonstrates Java entity beans for such a scenario:

SQL to create the tables for these entities:

```
CREATE TABLE `UserSSN` (
 `id` BIGINT not null auto_increment primary key,
 `socialSecurityNumber` VARCHAR(25)
);

CREATE TABLE `User` (
 `id` BIGINT not null auto_increment primary key,
 `username` VARCHAR(25),
 `userSSNID` BIGINT,
 UNIQUE (`userSSNID`),
 UNIQUE (`username`),
 FOREIGN KEY (`userSSNID`) REFERENCES UserSSN(`id`) ON DELETE CASCADE
);
```

Sample data:

```
MariaDB [todo]> select * from UserSSN;
+-----+-----+
| id | socialSecurityNumber |
+-----+-----+
| 1 | aaa-aa-aaaa |
| 2 | bbb-bb-bbbb |
| 3 | ccc-cc-cccc |
| 4 | ddd-dd-dddd |
+-----+-----+

MariaDB [todo]> select * from User;
+-----+-----+
| id | username | userSSNID |
+-----+-----+
| 1 | usera | 1 |
| 2 | userb | 2 |
| 3 | userc | 3 |
| 4 | userd | 4 |
+-----+-----+
```

### User Entity:

```
@Entity
public class User {

 @Id
```

```

@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;

private String username;

@OneToOne(optional=false)❶
@JoinColumn(name="userSSNID")❷
private UserSSN userSSN;

```

- ❶ The `@OneToOne` annotation tells JPA that a single instance of `UserSSN` is related to each `User` instance. The `optional` option tells JPA that this field is required, and an error is thrown if the database contains a row without a value for this column.
- ❷ The `@JoinColumn` annotation tells JPA which column on the `User` table to use when looking up the `UserSSN` row, in this case the column named `userSSNID` is used, which is the foreign key configured in the database.

### UserSSN Entity:

```

@Entity
public class UserSSN {

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;

private String socialSecurityNumber;

@OneToOne(optional=false, mappedBy="userSSN")❶
private User user;

```

- ❶ The `@OneToOne` annotation tells JPA that a single instance of `User` is related to each `UserSSN` instance. The `mappedBy` option is used because the `UserSSN` record doesn't have any direct reference to the `User`, instead it uses the name of the field `userSSN` on the `User` class itself.

The result of mapping this relationship on the entities is that when a `User` is retrieved from the database, JPA automatically uses an SQL `JOIN` to connect the `User` table to the `UserSSN` table and populates the `UserSSN` object on the `User` instance when it is returned to the application.

## Using a One-to-many Entity Relationship

Use the `@OneToMany` and `@ManyToOne` annotations to map a JPA relationship anytime two entities relate to each other such that one instance of one entity relates to potentially multiple instances of the other entity. An example of this is a database used to store user information, placing each user into a single group. For this, two entities are used: a `User` entity and a `UserGroup` entity. These two are related using the `@OneToMany` and `@ManyToOne` annotations, implying that many users belong to one group, and one group can be assigned to many users.

The following example shows Java entity beans for such a group scenario:

SQL to create the tables for these entities:

```

CREATE TABLE `UserGroup` (
 `id` BIGINT not null auto_increment primary key,
 `name` VARCHAR(25)
);

CREATE TABLE `User` (
 `id` BIGINT not null auto_increment primary key,
 `groupID` BIGINT,
 `username` VARCHAR(25),
 UNIQUE (`username`),
 FOREIGN KEY (`groupID`) REFERENCES UserGroup(`id`)
);

```

Sample data:

```

MariaDB [todo]> select * from UserGroup;
+----+-----+
| id | name |
+----+-----+
| 1 | Group A |
| 2 | Group B |
| 3 | Group C |
| 4 | Group D |
+----+-----+

MariaDB [todo]> select * from User;
+----+-----+-----+
| id | username | groupID |
+----+-----+-----+
| 1 | usera | 1 |
| 2 | userb | 2 |
| 3 | userc | 3 |
| 4 | userd | 4 |
+----+-----+-----+

```

### User Entity:

```

@Entity
public class User {

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;

 @ManyToOne①
 @JoinColumn(name="groupID") ②
 private UserGroup userGroup;
}

```

- ① The `@ManyToOne` annotation tells JPA that a single instance of `UserGroup` is potentially related to multiple `User` instances
- ② The `@JoinColumn` annotation tells JPA which column to use when joining to the `UserGroup` table, in this example `groupID`.

**UserGroup Entity:**

```

@Entity
public class UserGroup {

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;

 private String name;

 @OneToMany(mappedBy="userGroup") ①
 private Set<User> users;
}

```

- ① The `@OneToMany` annotation tells JPA that potentially multiple `User` instances can belong to a single `UserGroup` instance. The member variable on the `UserGroup` entity is `Set<User>`, which permits multiple `User` instances to be stored. The `mappedBy` option is used because the `UserGroup` table actually doesn't contain a reference to `user`, so JPA needs to use the attribute `userGroup` on the `User` entity to do the mapping when creating the query.

**Tuning the Performance of Loading Relationship Data**

Each relationship mapped with relationship JPA annotations requires extra processing to populate the relational data. While this is not typically a problem with small data sets or simple database schemas, it becomes cumbersome if developers are not careful. Nesting relationships or complex database schemas with lots of relationships can be especially harmful to performance, as sometimes multiple queries must be executed to retrieve necessary data, or complex joins must be used. It is also very easy to use JPA to retrieve more data than intended, especially if every relationship is mapped bidirectionally on both entities.

Consider a scenario where an application is storing automobile data. There are entities for `Make`, `Model`, `SubModel`, and `Car`. Each of these entities has a relationship to each other. For each `Make`, there are many `Model` instances, and for each `Model` there are many `SubModel` instances. Additionally, the `car` entity has `@ManyToOne` relationships to each of these entities.

**Make Entity:**

```

@Entity
public class Make {

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;

 private String name;

 @OneToMany(mappedBy="make")
 private Set<Model> models;
}

```

**Model Entity:**

```

@Entity
public class Model {

```

```

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;

private String name;

@ManyToOne
@JoinColumn(name="makeID")
private Make make;

@OneToMany(mappedBy="model")
private Set<SubModels> submodels;

```

**SubModel Entity:**

```

@Entity
public class SubModel {

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;

private String name;

@ManyToOne
@JoinColumn(name="modelID")
private Model model;

```

**Car Entity:**

```

@Entity
public class Car {

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;

@ManyToOne
@JoinColumn(name="makeID")
private Make make;

@ManyToOne
@JoinColumn(name="modelID")
private Model model;

@ManyToOne
@JoinColumn(name="subModelID")
private SubModel subModel;

private Long year;

```

If all the relationships between these entities are mapped using JPA annotations, anytime a Car is retrieved from the database, the Make, Model, and SubModel are also retrieved using joins.

However, because each of those entities has their interrelationships mapped, they also use the JPA annotations to populate their member variables. This means the SubModel retrieves its Model instances, the Model then retrieves its Make instances and all of its SubModel instances, the Make retrieves its Model instances. After all of this data is loaded by JPA into memory, most of the data in the database has been retrieved. This severely inhibits application performance with a large data set.

## Using Lazy-Loading to Improve JPA Performance

Even if one entity has a relationship mapped to another entity with JPA annotations, it is not always necessary to retrieve that related entity when loading the data for the first entity from the database. This depends on the business logic that processes the instance of the entity being loaded. For example, different screens in an application might require different amounts of information required to display or have different performance requirements.

In the example of the Car entity described previously, loading a specific Model when a Car is retrieved is probably required to display a specific car on screen, but loading all of the SubModel instances for that Model is probably unnecessary to display a specific car on the screen.

To alleviate this problem and improve entity loading performance, JPA provides functionality called *lazy loading*, or *lazy fetching*. With lazy loading, entities can map relationships, but only load those relationships when needed.

The behavior of whether or not JPA loads related entities is called the *fetch type*. There are two fetch types that can be used, *lazy* or *eager*. Eager fetching is the default setting, but can also be set explicitly. To enable lazy loading, developers must set the `FetchType` value on the JPA relationship annotation.

The following example shows a lazily loaded relationship:

```
@Entity
public class Make {

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;

 private String name;

 @OneToMany(mappedBy="make", fetch=FetchType.LAZY)
 private Set<Model> models;
```

After adding lazy loading, when a Make is retrieved from the database, JPA does not attempt to load all of its model instances. If later in the application logic, after the Make instance was retrieved from the database, other business logic attempts to call `getModels()`, JPA attempts to automatically call the database and populate the models. If the entity manager connection used to retrieve the Make instance is no longer active, a `LazyInitializationException` is thrown because JPA can no longer retrieve the models using the same connection.

## Using the Join Fetch Clause in JPQL Queries to Eagerly Fetch Related Entities

The `JOIN FETCH` clause in JPQL queries overrides lazy-loading behavior in entities and eagerly fetches the related entity data. `JOIN FETCH` is useful when developing queries because it separates management of the fetch behavior for each query. This allows developers to safely

include lazy fetching of related entity data by default, improving application performance by only eagerly fetching the related data as needed by including a `JOIN FETCH` clause in the JPQL query.

To use a `JOIN FETCH` clause, include the name of the JPA-mapped collection that JPA should fetch eagerly. A `JOIN FETCH` clause in a query overrides any `fetch` attribute specified on the JPA annotation that maps the relationship. When a `JOIN FETCH` is included, it results in JPA including a `JOIN` to the table mapped to the related entity in the SQL that is generated from the JPQL query. The following is an example of a query that uses a `JOIN FETCH` clause to load the set of `model` objects that are related to the `make` objects that JPA loads from the database:

```
TypedQuery<Make> query = em.createQuery("SELECT ma FROM Make ma JOIN FETCH
ma.models WHERE ma.id = :id" , Make.class);
```

## Demonstration: Configuring Entity Relationships

- Run the following command to prepare files used by this demonstration.

```
[student@workstation ~]$ demo configure-relationships setup
```

- Start JBDS and import the `configure-relationships` project.

This project is a simple web app using a JSF page backed by a stateful request-scoped EJB to display the employees in a department. It also uses CDI to inject the EJB responsible for querying the database.

- Inspect the `pom.xml` file, and observe the dependency on hibernate libraries for the JPA specification and entity manager classes.

```
<dependency>
<groupId>org.hibernate.javax.persistence</groupId>
<artifactId>hibernate-jpa-2.1-api</artifactId>
<scope>provided</scope>
</dependency>
<dependency>
<groupId>org.hibernate</groupId>
<artifactId>hibernate-entitymanager</artifactId>
<scope>provided</scope>
</dependency>
```

- Open a new terminal window and use the `mysql` client to query the `todo` database.

```
[student@workstation ~]$ mysql -hservices -utodo -ptodo
Welcome to the MariaDB monitor. Commands end with ; or \g.
Your MariaDB connection id is 11
Server version: 5.5.52-MariaDB MariaDB Server

Copyright (c) 2000, 2016, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
```

```
MariaDB [(none)]> use todo;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
```

Retrieve the data in the Department table.

```
MariaDB [todo]> select * from Department;
+----+-----+
| id | name |
+----+-----+
| 1 | Sales |
| 2 | Marketing |
| 3 | IT |
+----+-----+
3 rows in set (0.00 sec)
```

Retrieve the data in the Manager table.

```
MariaDB [todo]> select * from Manager;
+----+-----+
| id | name | departmentID |
+----+-----+-----+
| 1 | Bob | 1 |
| 2 | Jill | 2 |
| 3 | Sam | 3 |
+----+-----+
3 rows in set (0.00 sec)
```

The Manager table has a one-to-one relationship with the Department table.

Retrieve the data in the Employee table.

```
MariaDB [todo]> select * from Employee;
+----+-----+-----+
| id | name | departmentID |
+----+-----+-----+
| 1 | William | 1 |
| 2 | Rose | 1 |
| 3 | Pat | 1 |
| 4 | Rodney | 2 |
| 5 | Kim | 2 |
| 6 | Tom | 2 |
| 7 | Matt | 3 |
| 8 | George | 3 |
| 9 | Jean | 3 |
+----+-----+
9 rows in set (0.00 sec)
```

The Employee table has a many-to-one relationship with the Manager table.

5. Update the Employee entity class to include the necessary JPA annotations.

```
@Entity
public class Employee {

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;

 private String name;

 @ManyToOne
 @JoinColumn(name="departmentID")
 private Department department;
```

6. Update the Manager entity class to include the necessary JPA annotations.

```
@Entity
public class Manager {

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;

 private String name;

 @OneToOne
 @JoinColumn(name="departmentID")
 private Department department;
```

7. Update the Department entity class to include the necessary JPA annotations.

Eagerly fetch the employee records for a department so that the application will preload the employees when it retrieves the department records.

```
@Entity
public class Department {

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;

 private String name;

 @OneToOne(mappedBy="department")
 private Manager manager;

 @OneToMany(mappedBy="department", fetch=FetchType.EAGER)
 private Set<Employee> employees;
```

8. Start the local JBoss EAP server inside JBDS.

In the **Servers** tab in the bottom pane of JBDS, right-click the server **Red Hat JBoss EAP 7.0 [Stopped]** and click on the green button to start the server. Watch the **Console** until the server starts and you see the started message:

```
INFO [org.jboss.as] (Controller Boot Thread) WFLYSRV0025: JBoss EAP 7.0.0.GA
(WildFly Core 2.1.2.Final-redhat-1) started
```

9. Deploy the application to the local JBoss EAP server, and test it in a browser. Run the following command in a terminal window:

```
[student@workstation ~]$ cd /home/student/JB183/labs/configure-relationships
[student@workstation configure-relationships]$ mvn wildfly:deploy
```

Open <http://localhost:8080/configure-relationships/> in your browser, then test the application and make sure it is properly displaying the employees and manager of different departments.

10. Undeploy the application and stop the server.

```
[student@workstation configure-relationships]$ mvn wildfly:undeploy
```



## References

Further information is available in the JPA chapter of the *Development Guide* for Red Hat JBoss EAP 7 at  
<https://access.redhat.com/documentation/en/red-hat-jboss-enterprise-application-platform/>

## ► Guided Exercise

# Configuring Entity Relationships

In this exercise, you will configure entity relationships between multiple entities that are used in a JSF-based web application.

### Outcome

You should be able to correctly map one-to-one and many-to-one relationships using the necessary JPA annotations.

### Before You Begin

Open a terminal window on the **workstation** VM and run the following command to download the files required for this lab.

```
[student@workstation ~]$ lab entity-relationships setup
```

## Steps

- ▶ 1. Open JBDS and import the Maven project.
  - 1.1. Open JBDS by double-clicking the JBoss Developer Studio icon on the workstation desktop. Set the workspace to /home/student/JB183/workspace and click OK.
  - 1.2. In the JBDS menu, click **File > Import** to open the **Import** wizard.
  - 1.3. On the **Select** page, click **Maven > Existing Maven Projects**, and then click **Next**.
  - 1.4. In the **Maven projects** page, click **Browse** to open the **Select root folder** window. Navigate to the /home/student/JB183/labs/ directory. Select the **entity-relationships** folder, and then click **OK**.
  - 1.5. On the **Maven projects** page, click **Finish**.
  - 1.6. Watch the JBDS status bar to monitor the progress of the import operation. It may take a few minutes to download all the required dependencies.
- ▶ 2. Map the one-to-one relationships between the **User** and **Email** entities.
  - 2.1. Open the **Email** class by expanding the **entity-relationships** item in the **Project Explorer** tab in the left pane of JBDS, then click on **entity-relationships > Java Resources > src/main/java > com.redhat.training.model** and expand it. Double-click the **Email.java** file.

Add the **@OneToOne** JPA annotation to map the relationship to the **User** entity:

```
@Entity
public class Email {
```

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;

private String address;

//TODO map relationship
@OneToOne(mappedBy="email")
private User user;
```

Notice the `mappedBy` attribute used. This is because the column that dictates how to map user records to email records is stored in the user table, and is represented by the `email` variable on the `User` entity class.



### Note

If JBDS raises an error about the mapping to `User`, this is expected. This error is resolved in the next step.

- 2.2. Save your changes using `Ctrl+S`.
- 2.3. Open the `User` class by expanding the `entity-relationships` item in the `Project Explorer` tab in the left pane of JBDS, then click on `entity-relationships > Java Resources > src/main/java > com.redhat.training.model` and expand it. Double-click the `User.java` file.

Add the `@OneToOne` JPA annotation to map the relationship to the `Email` entity:

```
@Entity
public class User {

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;

 private String name;

 //TODO map relationship
 @OneToOne
 @JoinColumn(name="emailID")
 private Email email;
```

Notice the `@JoinColumn` annotation. This tells JPA which column to use when joining the `User` table to the `Email` table. In this case, it uses the `emailID` column.

- 2.4. Save your changes using `Ctrl+S`.
  3. Map the one-to-many relationship between the `UserGroup` and `User` entities.
    - 3.1. Open the `UserGroup` class by expanding the `entity-relationships` item in the `Project Explorer` tab in the left pane of JBDS, then click on `entity-relationships > Java Resources > src/main/java > com.redhat.training.model` and expand it. Double-click the `UserGroup.java` file.
- Add the `@OneToMany` JPA annotation to map the relationship to the `UserGroup` entity:

```

@Entity
public class UserGroup {

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;

 private String name;

 @OneToMany(mappedBy="userGroup")
 private Set<User> users;
}

```

Notice the `mappedBy` attribute. This is used because the column to map user records to user-group records is stored in the user table, and is represented by the `userGroup` variable on the `User` entity class.



### Note

If JBDS raises an error about the mapping to `User`, this is expected. This error is resolved in the next step.

3.2. Save your changes using `Ctrl+S`

3.3. Open the `User` class by expanding the `entity-relationships` item in the `Project Explorer` tab in the left pane of JBDS, then click on `entity-relationships > Java Resources > src/main/java > com.redhat.training.model` and expand it. Double-click the `User.java` file.

Add the `@ManyToOne` JPA annotation to map the relationship to the `UserGroup` entity:

```

@Entity
public class User {

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;

 private String name;

 //TODO map relationship
 @OneToOne
 @JoinColumn(name="emailID")
 private Email email;

 //TODO map relationship
 @ManyToOne
 @JoinColumn(name="groupID")
 private UserGroup userGroup;
}

```

Notice the `@JoinColumn` annotation. This tells JPA which column to use when joining the `User` table to the `UserGroup` table. In this case, it uses the `groupID` column.

- 3.4. Save your changes using **Ctrl+S**.
- 4. Start EAP by selecting the **Servers** tab in the bottom pane of JBDS. Right-click the server **Red Hat JBoss EAP 7.0 [Stopped]**, and click the green button to start the server. Watch the **Console** until the server starts and displays the message:

```
INFO [org.jboss.as] (Controller Boot Thread) WFLYSRV0025: JBoss EAP 7.0.0.GA
(WildFly Core 2.1.2.Final-redhat-1) started
```

- 5. Deploy the application on JBoss EAP using Maven by running the following commands:

```
[student@workstation ~]$ cd /home/student/JB183/labs/entity-relationships
[student@workstation entity-relationships]$ mvn wildfly:deploy
```

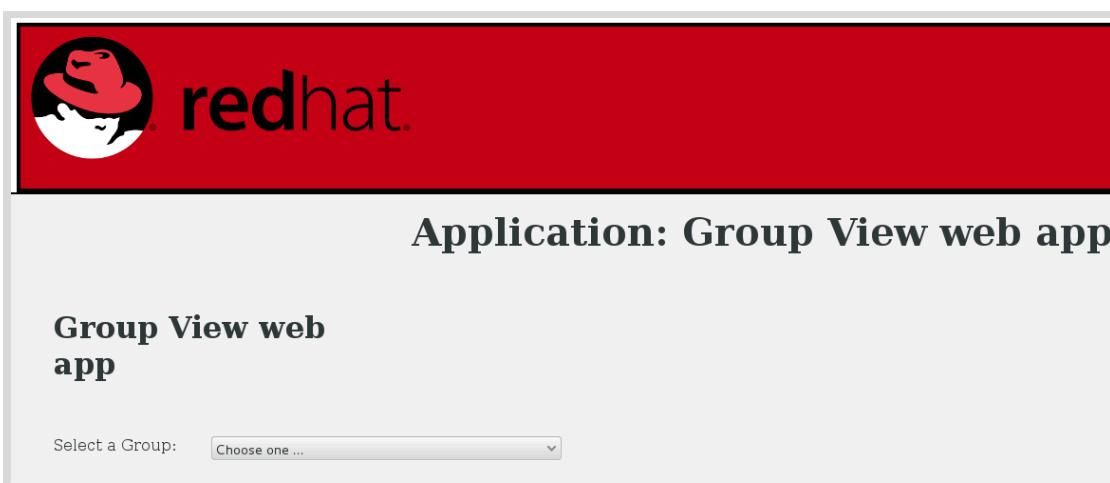
Once complete, **BUILD SUCCESS** displays, as shown in the following example:

```
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 17.116 s
[INFO] Finished at: 2016-12-01T07:26:55-05:00
[INFO] Final Memory: 35M/210M
[INFO] -----
```

Validate the deployment in the server log shown in the **Console** tab in JBDS. When the app is deployed, the following appears in the log:

```
INFO [org.jboss.as.server] (management-handler-thread - 9) WFLYSRV0010: Deployed
"entity-relationships.war" (runtime-name : "entity-relationships.war")
```

- 6. Test the application in a browser.
- 6.1. Open <http://localhost:8080/entity-relationships> in a browser on the workstation VM.



**Figure 5.1: Application home page**

- 6.2. Choose a group from the dropdown to view.

The page updates with a stack trace because of a `LazyInitializationException`:

```

Error processing request
Context Path:/entity-relationships
Servlet Path:/index.jsf
Path Info:null
Query String:null
Stack Trace
javax.servlet.ServletException: org.hibernate.LazyInitializationException: failed to lazily initialize a collection of role: com.redhat.training.model.UserGroup.users, initialize proxy - no Session
javax.faces.webapp.FacesServlet.service(FacesServlet.java:671)
io.undertow.servlet.handlers.ServletHandler.handleRequest(ServletHandler.java:85)
io.undertow.servlet.handlers.security.SerlvetSecurityRoleHandler.handleRequest(SerlvetSecurityRoleHandler.java:62)
io.undertow.servlet.handlers.ServletDispatchingHandler.handleRequest(SerlvetDispatchingHandler.java:36)
org.wildfly.extension.undertow.security.SecurityContextAssociationHandler.handleRequest(SecurityContextAssociationHandler.java:78)
io.undertow.server.handlers.PredicateHandler.handleRequest(PredicateHandler.java:43)
io.undertow.servlet.handlers.security.SSLInformationAssociationHandler.handleRequest(SSLInformationAssociationHandler.java:131)
io.undertow.servlet.handlers.security.SerlvetAuthenticationCallHandler.handleRequest(SerlvetAuthenticationCallHandler.java:57)
io.undertow.server.handlers.PredicateHandler.handleRequest(PredicateHandler.java:43)
io.undertow.security.handlers.AbstractConfidentialityHandler.handleRequest(AbstractConfidentialityHandler.java:46)
io.undertow.servlet.handlers.security.SerlvetConfidentialityConstraintHandler.handleRequest(SerlvetConfidentialityConstraintHandler.java:64)

```

**Figure 5.2: Application error response**

Notice the first error message:

```
org.hibernate.LazyInitializationException: failed to lazily initialize a collection of role: com.redhat.training.model.UserGroup.users, could not initialize proxy - no Session
```

This error was caused by the fact that the JSF page tried to load the set of users from a group that was loaded from the database, but the users were not eagerly loaded. Therefore, the JPA session was already closed when the JSF page attempted the call to `getUsers()`, causing the `LazyInitializationException`.

This error can be fixed with eager fetching.

- ▶ 7. Update the `UserGroup` entity to use eager fetching on this set of users.
    - 7.1. Open the `UserGroup` class by expanding the `entity-relationships` item in the `Project Explorer` tab in the left pane of JBDS, then click `entity-relationships > Java Resources > src/main/java > com.redhat.training.model` and expand it. Double-click the `UserGroup.java` file.
- Add the `fetch` attribute to the JPA annotation and set the fetch type to `FetchType.EAGER` to eagerly load the relationship to the `User` entity:

```

@Entity
public class UserGroup {

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;

 private String name;

 @OneToMany(mappedBy="userGroup", fetch=FetchType.EAGER)
 private Set<User> users;
}

```

- 7.2. Save your changes using `Ctrl+S`.

- 8. Redeploy the application on JBoss EAP using Maven by running the following commands:

```
[student@workstation entity-relationships]$ mvn wildfly:deploy
```

Once complete, you should see **BUILD SUCCESS** as shown in the following example:

```
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 17.116 s
[INFO] Finished at: 2016-12-01T07:26:55-05:00
[INFO] Final Memory: 35M/210M
[INFO] -----
```

Validate the deployment in the server log shown in the **Console** tab in JBDS. When your app has been deployed successfully, this appears in the log::

```
INFO [org.jboss.as.server] (management-handler-thread - 9) WFLYSRV0016: Replaced deployment "entity-relationships.war" with deployment "entity-relationships.war"
```

- 9. Test the application in a browser.

- 9.1. Open <http://localhost:8080/entity-relationships> in a browser on the **workstation** VM.
- 9.2. Choose a group and view its users. The app displays users and email addresses of the group.

## Group View web app

Select a Group:

Employees:

- Name:Rodney  
Email:research@example.com
- Name:Kim  
Email:ceo@example.com
- Name:Tom  
Email:problems@example.com

**Figure 5.3: Correct application response**

- 10. Update the **UserGroup** entity to use lazy fetching on this set of users and override this behavior using a **JOIN FETCH** in the query.

- 10.1. In the `UserGroup` entity class in the `com.redhat.training.model`, update the `fetch` attribute of the JPA annotation and set the fetch type to `FetchType.LAZY` to lazily load the relationship to the `User` entity:

```
@Entity
public class UserGroup {

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;

 private String name;

 @OneToMany(mappedBy="userGroup", fetch=FetchType.LAZY)
 private Set<User> users;
```

- 10.2. Save your changes using `Ctrl+S`.

- 10.3. Update the `UserBean` EJB in the `com.redhat.training.ejb` package to use a `JOIN FETCH` to eagerly fetch the set of users for a group.

```
@Stateless
public class UserBean {

 @Inject
 private EntityManager em;

 public Set<UserGroup> getAllUserGroups(){
 TypedQuery<UserGroup> query = em.createQuery("SELECT g FROM UserGroup g JOIN
 FETCH g.users" , UserGroup.class);

 return new HashSet<UserGroup>(query.getResultList());
 }
}
```

- 10.4. Save your changes using `Ctrl+S`.

- 11. Redeploy the application on JBoss EAP using Maven by running the following commands:

```
[student@workstation entity-relationships]$ mvn wildfly:deploy
```

Once complete, you should see `BUILD SUCCESS` as shown in the following example:

```
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 17.116 s
[INFO] Finished at: 2016-12-01T07:26:55-05:00
[INFO] Final Memory: 35M/210M
[INFO] -----
```

Validate the deployment in the server log shown in the **Console** tab in JBDS. When your app has been deployed successfully, this appears in the log:

```
INFO [org.jboss.as.server] (management-handler-thread - 9) WFLYSRV0016: Replaced deployment "entity-relationships.war" with deployment "entity-relationships.war"
```

► **12.** Test the application in a browser.

- 12.1. Open <http://localhost:8080/entity-relationships> in a browser on the workstation VM.
- 12.2. Choose a group and view its users. The application displays the users and email addresses of the group without any errors.

► **13.** Undeploy the application and stop JBoss EAP.

- 13.1. Run the following command to undeploy the application:

```
[student@workstation entity-relationships]$ mvn wildfly:undeploy
```

- 13.2. To close the project, right-click on the `entity-relationships` project in the **Project Explorer**, and select **Close Project**.
- 13.3. Right-click on the Red Hat JBoss EAP 7.0 server in the **JBDS Servers** tab and click **Stop**.

This concludes the guided exercise.

# Describing Many-to-many Entity Relationships

---

## Objective

After completing this section, students should be able to describe many-to-many entity relationships.

## Understanding Many-to-many Relationships

A many-to-many relationship occurs when each row in one table has multiple related rows in another table, and vice versa. An example of this type of relationship is assigning workers to projects. Each worker can be assigned to one or more projects and each project can have multiple workers. This type of relationship is complicated to represent in a relational database because unlike a one-to-many relationship, a single column on one of the tables cannot be used to represent the relationship.

To represent a many-to-many relationship in a relational database, an intermediary table known as a *cross-table* or *join-table* must be used. A join-table is an intermediary table where each row in the table represents combinations of two IDs, one from each of the tables.

An example of a many-to-many relationship in an enterprise application is a database that stores school class enrollment data. A `Student` table is used to store student records, and a `Class` table is used to store classes. A `StudentXClass` join-table is used to relate these two tables, and to represent which students are in which classes. Using these three tables, it is possible to find the students in a given class, as well as the classes for a given student by joining to the `StudentXClass` table.

Here are the Java entity beans for such a class enrollment application:

The SQL to create the tables for these entities:

```
CREATE TABLE `Student` (
 `id` BIGINT not null auto_increment primary key,
 `name` VARCHAR(50)
);

CREATE TABLE `Class` (
 `id` BIGINT not null auto_increment primary key,
 `name` VARCHAR(50)
);

CREATE TABLE `StudentXClass` (
 `id` BIGINT not null auto_increment primary key,
 `studentID` BIGINT not null,
 `classID` BIGINT not null,
 FOREIGN KEY (`studentID`) REFERENCES Student(`id`),
 FOREIGN KEY (`classID`) REFERENCES Class(`id`)
);
```

### Student Entity:

```
@Entity
public class Student {

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;

 private String name;

 private Set<Class> classes;
```

**Class Entity:**

```
@Entity
public class Class {

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;

 private String name;

 private Set<Student> students;
```

Notice that there is not an entity for the `StudentXClass` join-table. This is because Java objects don't require an intermediary object to represent a many-to-many relationship between two other objects. Creating a `Set` of each of the entities as a member variable of the other, as seen in the previous example, is enough to create the many-to-many relationship between the two objects.

## Using JPA Annotations to Map Many-to-many Relationships

Instead of using a single `@JoinColumn` JPA, use a `@JoinTable` annotation and two `@JoinColumn` annotations to map a many-to-many relationship. The `@JoinTable` annotation is the bridge that allows JPA to use the join-table to populate the relationships between two objects, and the two `@JoinColumn` annotations are used to define how to join the join-table back to the entity table, as well as to the related entity table.

When specifying the `@JoinTable` annotation, the `name`, `joinColumns`, and `inverseJoinColumns` attributes are required and used in the following way:

```
@JoinTable(
 name="$JOIN_TABLE_NAME",
 joinColumns=@JoinColumn(name="$JOIN_TABLE_COLUMN_NAME"),
 inverseJoinColumns=@JoinColumn(name="$JOIN_TABLE_COLUMN2_NAME"))
private Set<RelatedEntity> entities;
```

**name**

The name of the join-table to be used by JPA when populating the relationship.

**joinColumns**

The column of the join-table to use to join back to the entity class. This attribute accepts an instance of `@JoinColumn`, which is defined with a `name` attribute that maps to the column name on the join-table.

**inverseJoinColumns**

The column of the join-table to use to join from the join-table to the related entity class. This attribute accepts an instance of `@JoinColumn`, defined with a `name` attribute that maps to the column name on the join-table.

A final consideration when mapping many-to-many relationships in JPA entities is that each of the relationships only needs its join-table mapped on one side of the relationship. Developers generally refer to this side as the *owner* or *parent* of the relationship. Once a parent is defined, the related entity or *child* entity can use the `mappedBy` attribute on the `@ManyToMany` annotation to reference the variable on the owning entity which maps the relationship.

**Important**

Only one side of a JPA relationship can be mapped as the owner. If both entities have the join-table mapped and neither uses `mappedBy`, JPA tries to persist duplicate rows in the join-table.

Continuing the class enrollment example from the previous section, the correctly annotated JPA entity classes `Student` and `Class` matches the following:

**Student Entity:**

```
@Entity
public class Student {

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;

 private String name;

 @ManyToMany(1)
 @JoinTable(2
 name="StudentXClass", 3
 joinColumns=@JoinColumn(name="studentID"), 4
 inverseJoinColumns=@JoinColumn(name="classID")) 5
 private Set<Class> classes;
```

- ➊ Mark this relationship as a many-to-many relationship with the `@ManyToMany` annotation.
- ➋ Specify the join-table to use when populating the relationship with the `@JoinTable` annotation.
- ➌ Set the `name` attribute to the table name of the join-table.
- ➍ Specify the `@JoinColumn` to use to join the join-table back to the `Student` table, in this case `studentID`.
- ➎ Specify the `@JoinColumn` to use to join the join-table back to the related `Class` table, in this case `classID`. The `inverseJoinColumn` will always be a column on the related entity.

**Class Entity:**

```
@Entity
public class Class {

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;

 private String name;

 @ManyToMany(mappedBy="classes")①
 private Set<Student> students;
```

- ① Specify the `Class` entity as the child of the relationship with the `Student` entity by using the `mappedBy` attribute on the `@ManyToMany` annotation. Reference the variable on the `Student` object that can be used to map the relationship; in this case, `classes`.

**References**

Further information is available in the JPA chapter of the *Development Guide* for Red Hat JBoss EAP 7; at <https://access.redhat.com/documentation/en/red-hat-jboss-enterprise-application-platform/>

## ► Quiz

# Describing Many-to-many Entity Relationships

Choose the correct answer(s) to the following questions:

► 1. Which three JPA annotations are required to fully map a many-to-many relationship?

(Choose three.):

- a. @JoinColumn
- b. @ManyToOne
- c. @JoinTable
- d. @OneToMany
- e. @ManyToMany

► 2. Which scenario describes a many-to-many relationship?

- a. Data representing which animals are in which zoos at any given time.
- b. Data representing children and their biological mothers.
- c. Data representing professional athletes and their current teams.
- d. Data representing research papers and every journal that published them.

► 3. SQL:

```
CREATE TABLE `Employee` (
 `id` BIGINT not null auto_increment primary key,
 `name` VARCHAR(50)
);

CREATE TABLE `Project` (
 `id` BIGINT not null auto_increment primary key,
 `name` VARCHAR(50)
);

CREATE TABLE `EmployeeXProject` (
 `id` BIGINT not null auto_increment primary key,
 `employeeID` BIGINT not null,
 `projectID` BIGINT not null,
 FOREIGN KEY (`employeeID`) REFERENCES Employee(`id`),
 FOREIGN KEY (`projectID`) REFERENCES Project(`id`)
);
```

- 4. Given the structure of the entity classes in the previous example, which JPA annotation should be used on the Set of Employee instances to make the Project entity the child of the relationship?

- a. @ManyToMany(mappedBy="projects")
- b. @JoinTable(name="Project",  
joinColumns=@JoinColumn(name="id"),  
inverseJoinColumns=@JoinColumn(name="id"))
- c. @JoinTable(name="EmployeeXProject",  
joinColumns=@JoinColumn(name="employeeID"),  
inverseJoinColumns=@JoinColumn(name="projectID"))
- d. @ManyToOne

## ► Solution

# Describing Many-to-many Entity Relationships

Choose the correct answer(s) to the following questions:

► 1. Which three JPA annotations are required to fully map a many-to-many relationship?

(Choose three.):

- a. @JoinColumn
- b. @ManyToOne
- c. @JoinTable
- d. @OneToMany
- e. @ManyToMany

► 2. Which scenario describes a many-to-many relationship?

- a. Data representing which animals are in which zoos at any given time.
- b. Data representing children and their biological mothers.
- c. Data representing professional athletes and their current teams.
- d. Data representing research papers and every journal that published them.

► **3. SQL:**

```
CREATE TABLE `Employee` (
 `id` BIGINT not null auto_increment primary key,
 `name` VARCHAR(50)
);

CREATE TABLE `Project` (
 `id` BIGINT not null auto_increment primary key,
 `name` VARCHAR(50)
);

CREATE TABLE `EmployeeXProject` (
 `id` BIGINT not null auto_increment primary key,
 `employeeID` BIGINT not null,
 `projectID` BIGINT not null,
 FOREIGN KEY (`employeeID`) REFERENCES Employee(`id`),
 FOREIGN KEY (`projectID`) REFERENCES Project(`id`)
);
```

- 4. Given the structure of the entity classes in the previous example, which JPA annotation should be used on the Set of Employee instances to make the Project entity the child of the relationship?

- a. @ManyToMany(mappedBy="projects")
- b. @JoinTable(name="Project",  
joinColumns=@JoinColumn(name="id"),  
inverseJoinColumns=@JoinColumn(name="id"))
- c. @JoinTable(name="EmployeeXProject",  
joinColumns=@JoinColumn(name="employeeID"),  
inverseJoinColumns=@JoinColumn(name="projectID"))
- d. @ManyToMany

## ► Lab

# Managing Entity Relationships

In this lab, you will update entity classes with the appropriate JPA annotations to map relationships between entities.

### Outcome

You should be able to map a one-to-one and a one-to-many relationship between entities.

### Before You Begin

Open a terminal window on the workstation VM and run the following command to download the files that are required for this lab.

```
[student@workstation ~]$ lab manage-relationships setup
```

1. Open JBDS, and import the `manage-relationships` project located in the `/home/student/JB183/labs/manage-relationships` directory.
2. Map the one-to-one relationship between `Teacher` and `Classroom` using JPA annotations.
3. Map the one-to-many relationship between the `Classroom` and `Student` entities and eagerly fetch the list of students when a `Classroom` object is loaded from the database.
4. Start JBoss EAP from within JBDS.
5. Build and deploy the application to JBoss EAP using Maven.
6. Test the application in a browser.
  - 6.1. Open Firefox on the workstation VM, and navigate to `http://localhost:8080/manage-relationships` to access the application.
  - 6.2. On the application screen, update the classroom to the different options. Make sure that the teacher and student data is populating correctly for each classroom.This verifies that JPA is able to use the annotations provided to map the relationships between the entities and populate the relationship data correctly.
7. Open a new terminal window and run the following command to grade the lab:

```
[student@workstation ~]$ lab manage-relationships grade
```

The grading script should report `SUCCESS`. If there is a failure, check the errors and fix them until you see a `SUCCESS` message.

8. Clean up.
  - 8.1. Undeploy the application from JBoss EAP using Maven with the following commands:

```
[student@workstation ~]$ cd /home/student/JB183/labs/manage-relationships
[student@workstation manage-relationships]$ mvn wildfly:undeploy
```

- 8.2. Right-click on the **manage-relationships** project in the **Project Explorer**, and select **Close Project** to close this project.
- 8.3. Right-click on the Red Hat JBoss EAP 7.0 server in the JBDS **Servers** tab and click **Stop**.

This concludes the lab.

## ► Solution

# Managing Entity Relationships

In this lab, you will update entity classes with the appropriate JPA annotations to map relationships between entities.

### Outcome

You should be able to map a one-to-one and a one-to-many relationship between entities.

### Before You Begin

Open a terminal window on the workstation VM and run the following command to download the files that are required for this lab.

```
[student@workstation ~]$ lab manage-relationships setup
```

1. Open JBDS, and import the `manage-relationships` project located in the `/home/student/JB183/labs/manage-relationships` directory.
  - 1.1. To open JBDS, double-click the JBoss Developer Studio icon on the workstation desktop. Leave the default workspace (`/home/student/JB183/workspace`) and click **OK**.
  - 1.2. In the JBDS menu, click **File > Import** to open the **Import** wizard.
  - 1.3. On the **Select** page, click **Maven > Existing Maven Projects**, and then click **Next**.
  - 1.4. In the **Maven projects** page, click **Browse** to open the **Select root folder** window. Navigate to the `/home/student/JB183/labs/` directory. Select the `manage-relationships` folder, and then click **OK**.
  - 1.5. On the **Maven projects** page, click **Finish**.
  - 1.6. Watch the JBDS status bar to monitor the progress of the import operation. It may take a few minutes to download all the required dependencies.
2. Map the one-to-one relationship between `Teacher` and `Classroom` using JPA annotations.
  - 2.1. Open the `Classroom` class by expanding the `manage-relationships` item in the **Project Explorer** tab in the left pane of JBDS, then click on `manage-relationships > Java Resources > src/main/java > com.redhat.training.model` and expand it. Double-click the `Classroom.java` file.

Add the `@OneToOne` JPA annotation to map the relationship to the `Teacher` entity:

```
@Entity
public class Classroom {

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;
```

```

private String name;

//TODO map relationship
@OneToOne(mappedBy="classroom")
private Teacher teacher;

```

Use the `mappedBy` attribute to map classroom records to teacher records using the column on the teacher table, which is represented by the `classroom` variable on the `Teacher` entity class.



### Note

JBDS should raise an error about mapping to Teacher, and this is expected. This error is resolved in the next step.

- 2.2. Save your changes using **Ctrl+S**.
- 2.3. To open the `Teacher` class, expand the `manage-relationships` item in the `Project Explorer` tab in the left pane of JBDS. Click on `manage-relationships > Java Resources > src/main/java > com.redhat.training.model` and expand it. Double-click the `Teacher.java` file.

Add the `@OneToOne` JPA annotation to map the relationship to the `Classroom` entity:

```

@Entity
public class Teacher {

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;

 private String name;

 @OneToOne
 @JoinColumn(name="classroomID")
 private Classroom classroom;
}

```

Use the `@JoinColumn` annotation to tell JPA which column to use when joining the `Teacher` table to the `Classroom` table, in this case, the `classroomID` column.

- 2.4. Save your changes using **Ctrl+S**.
  3. Map the one-to-many relationship between the `Classroom` and `Student` entities and eagerly fetch the list of students when a `Classroom` object is loaded from the database.
    - 3.1. To open the `Classroom` class, expand the `manage-relationships` item in the `Project Explorer` tab in the left pane of JBDS. Click on `manage-relationships > Java Resources > src/main/java > com.redhat.training.model` and expand it. Double-click the `Classroom.java` file.
- Add the `@OneToMany` JPA annotation to map the relationship to the `UserGroup` entity:

```

@Entity
public class Classroom {

```

```

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;

private String name;

//TODO map relationship
@OneToOne(mappedBy="classroom")
private Teacher teacher;

//TODO map relationship
@OneToMany(mappedBy="classroom")
private Set<Student> students;

```

Use the `mappedBy` attribute to reference the column mapping student records to classroom records, located on the student table, and represented by the `classroom` variable on the `Student` entity class.



### Note

If JBDS raises an error about the mapping to `Student`, this is expected. This error is resolved in the next step.

- 3.2. Update the `@OneToMany` relationship to be eagerly loaded using the `fetch` attribute:

```

//TODO map relationship
@OneToMany(mappedBy="classroom", fetch=FetchType.EAGER)
private Set<Student> students;

```

- 3.3. Save your changes using **Ctrl+S**

- 3.4. Open the `Student` class by expanding the `manage-relationships` item in the Project Explorer tab in the left pane of JBDS, then click on `manage-relationships` > Java Resources > `src/main/java` > `com.redhat.training.model` and expand it. Double-click the `Student.java` file.

Add the `@ManyToOne` JPA annotation to map the relationship to the `Classroom` entity:

```

@Entity
public class Student {

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;

 private String name;

 @ManyToOne
 @JoinColumn(name="classroomID")
 private Classroom classroom;
}

```

Be sure to use the `@JoinColumn` annotation to tell JPA which column to use when joining the `Student` table to the `Classroom` table. In this case, use the `classroomID` column.

3.5. Save your changes using **Ctrl+S**.

**4.** Start JBoss EAP from within JBDS.

Select the **Servers** tab in JBDS. Right-click on the server entry **Red Hat JBoss EAP 7.0 [Stopped]** and click the green **Start** option to start the server. Watch the **Console** tab of JBDS until the server starts and you see the message:

```
INFO [org.jboss.as] (Controller Boot Thread) WFLYSRV0025: JBoss EAP 7.0.2.GA
(WildFly Core 2.1.8.Final-redhat-1) started
```

**5.** Build and deploy the application to JBoss EAP using Maven.

Open a new terminal, and change directory to the `/home/student/JB183/labs/manage-relationships` folder.

```
[student@workstation ~]$ cd /home/student/JB183/labs/manage-relationships
```

Build and deploy the EJB to JBoss EAP by running the following command:

```
[student@workstation manage-relationships]$ mvn clean wildfly:deploy
```

A successful build displays the following output:

```
[student@workstation manage-relationships]$ mvn clean package wildfly:deploy
...
[INFO] [INFO] <<< wildfly-maven-plugin:1.0.2.Final:deploy (default-cli) < package
@ manage-relationships <<<
...
[INFO] --- maven-war-plugin:2.1.1:war (default-war) @ manage-relationships ---
...
[INFO] --- wildfly-maven-plugin:1.0.2.Final:deploy (default-cli) @ manage-relationships
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
```

**6.** Test the application in a browser.

6.1. Open Firefox on the **workstation** VM, and navigate to `http://localhost:8080/manage-relationships` to access the application.

6.2. On the application screen, update the classroom to the different options. Make sure that the teacher and student data is populating correctly for each classroom.

This verifies that JPA is able to use the annotations provided to map the relationships between the entities and populate the relationship data correctly.

**7.** Open a new terminal window and run the following command to grade the lab:

```
[student@workstation ~]$ lab manage-relationships grade
```

The grading script should report **SUCCESS**. If there is a failure, check the errors and fix them until you see a **SUCCESS** message.

**8.** Clean up.

8.1. Undeploy the application from JBoss EAP using Maven with the following commands:

```
[student@workstation ~]$ cd /home/student/JB183/labs/manage-relationships
[student@workstation manage-relationships]$ mvn wildfly:undeploy
```

- 8.2. Right-click on the `manage-relationships` project in the **Project Explorer**, and select **Close Project** to close this project.
- 8.3. Right-click on the Red Hat JBoss EAP 7.0 server in the **JBDS Servers** tab and click **Stop**.

This concludes the lab.

# Summary

---

In this chapter, you learned:

- Relationships between JPA entity classes are mapped using annotations.
- There are three main types of relationships between entities: one-to-one, one-to-many, and many-to-many. JPA provides annotations to map each of these relationship types.
- When mapping relationships between entities, it is important to consider performance implications of making JPA load related entities. When possible, leverage lazy-loading to avoid slower performance when related entities are not needed.
- JPA provides the `fetch` attribute, which can be set to `FetchType.EAGER` or `FetchType.LAZY` when mapping related entities. If lazy fetching is used for relationship mapping, then attempting to load that relationship data after the initial entity manager session results in a `LazyInitializationException`.
- A many-to-many relationship requires the use of a join-table. JPA provides the `@JoinTable` annotation for this purpose, and leverages multiple `@JoinColumn` to map many-to-many relationship between two entities.
- When a developer maps a many-to-many relationship between two entities, only one side can be mapped using the `@JoinTable`: the owning side. The other side of the relationship, known as the child side, must be mapped using the `mappedBy` attribute on the `@ManyToMany` annotation, which references a variable on the owning entity.



## Chapter 6

# Creating REST Services

### Overview

**Goal** Create REST APIs using the JAX-RS specification.

**Objectives**

- Describe web services concepts and list types of web services.
- Create a REST service using the JAX-RS specification.
- Create a client application that can invoke REST APIs remotely.

**Sections**

- Describing Web Services Concepts (and Quiz)
- Creating REST Services with JAX-RS (and Guided Exercise)
- Consuming a REST Service (and Quiz)

**Lab** Creating REST Services

# Describing Web Services Concepts

---

## Objective

After completing this section, students should be able to describe web services concepts and types.

## Web Services

Web services expose standardized communication for interoperability between application components over HTTP. By abstracting applications into individual components that communicate across web services, each system becomes loosely coupled to each other. This separation provides a greater ability to modify applications or integrate new systems into the application. Using a standard format for data transfer, such as JSON or XML, allows applications that consume web services to require only the ability to make an HTTP request to the service and to process the service's response.

In recent years, enterprise applications built on a foundation of web services have grown in popularity. One of the primary reasons for this increase in adoption is the need for applications to support multiple devices, such as desktop and mobile, while reducing development time to support diverse applications. By abstracting out the device-specific presentation layer, the data layer becomes a service layer. This separation allows organizations to quickly develop applications on a variety of platforms while reusing a shared back end built with web services. This approach reduces time to develop applications and to maintain changes to the back-end by not requiring work to be repeated across all supported platforms.

For example, consider a web application for a bank. The business wants to expand into the mobile market with a brand new mobile application. The developers' first step is to expose an API to access the application data. By exposing the bank's back end with a web services layer, the front end of the web application separates from the business logic of the application. As a result, the developers of the bank's application can create a mobile application front-end using web services without affecting the existing front-end application. Another benefit to exposing the services layer is that other front-end applications or web components needing the application's data can also call the same service endpoints. The following graphic demonstrates this architecture:

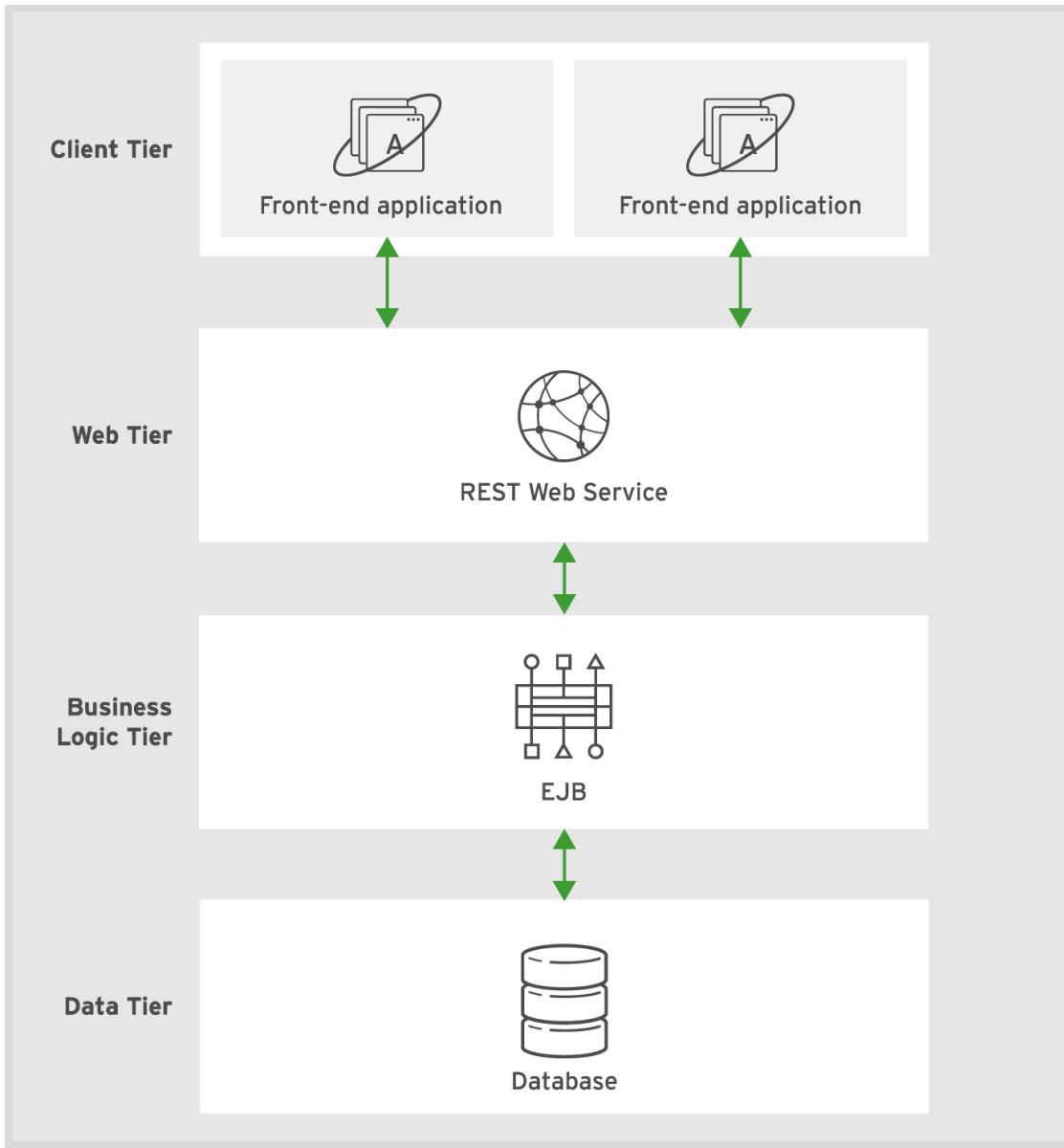


Figure 6.1: Web service application architecture

## Types of Web Services

There are two implementations of web services that are discussed in this course:

- JAX-RS *RESTful* Web Services
- JAX-WS Web Services

Both of these implementations provide the same benefits of using web services, such as loose coupling and standardized protocols, however JAX-WS and JAX-RS differ in a number of important ways that must be carefully considered before deciding which approach to use.



### Note

This course primarily uses JAX-RS. Discussions about JAX-WS are included to provide contrast to RESTful web services.

## JAX-RS

JAX-RS is the Java API for creating lightweight RESTful web services. In Red Hat JBoss EAP 7, the implementation of JAX-RS is *RESTEasy*, which is fully compliant with the JSR-311 specification entitled Java API for RESTful Web Services 2.0 and provides additional features for efficient development of REST services.

Developers can build RESTEasy web services by using annotations to mark certain classes and methods as endpoints. Each endpoint represents a URL that a client application can call and, depending on the type of annotation, specify the type of HTTP request.

In contrast with other approaches to web services, RESTful web services can use a smaller message format, such as JSON, compared to XML and others that create more overhead for each request. Each endpoint can be annotated to determine both the format of the data received and the format of the data returned to the client. Also, RESTful web services do not require use of a WSDL or anything similar to what is required when consuming JAX-WS services. This makes consuming RESTful web services much simpler, as consumers can simply make requests to individual endpoints in a service.

## JAX-WS

JAX-WS is the Java API for XML-based web services using the Simple Object Access Protocol (SOAP). *JBossWS* is the JSR-224 Java API for XML-based Web Services 2.2 specification compliant implementation for JAX-WS in Red Hat JBoss EAP 7.

To define a standard protocol for communication between applications, JAX-WS services use an XML definition file written using Web Services Description Language (WSDL). In many ways, WSDL simplifies the creation of web services by allowing an IDE, such as JBoss Developer Studio, to use the service definition to create clients that can interact with the service automatically. This service definition, however, does require more maintenance for the developers of the service. JAX-WS services also require clients and consumers to make more formal requests compared to JAX-RS, which can make requests to individual endpoints simply over HTTP.



### References

#### JSR-311 Java API for RESTful Web Services 2.0

<http://www.jcp.org/en/jsr/detail?id=311>

#### JSR-224 Java API for XML-based Web Services 2.2

<http://www.jcp.org/en/jsr/detail?id=224>



### References

Further information is available in the *Developing Web Services Guide* for Red Hat JBoss EAP 7; at

[https://access.redhat.com/documentation/en-us/red\\_hat\\_jboss\\_enterprise\\_application\\_platform/](https://access.redhat.com/documentation/en-us/red_hat_jboss_enterprise_application_platform/)

## ► Quiz

# Web Services

Match the items below to their counterparts in the table.

A lightweight, readable data format used by RESTful web services.

An XML format that describes the endpoints for a SOAP service.

The SOAP service EAP implementation.

The annotation-based web service implementation.

The specification for JAX-RS.

The specification for JAX-WS.

Term	Definition
WSDL	
JSON	
JSR-311 Java API for RESTful Web Services 2.0	
JSR-224 Java API for XML-based Web Services 2.2	
RESTEasy	
JBossWS	

## ► Solution

# Web Services

Match the items below to their counterparts in the table.

Term	Definition
WSDL	An XML format that describes the endpoints for a SOAP service.
JSON	A lightweight, readable data format used by RESTful web services.
JSR-311 Java API for RESTful Web Services 2.0	The specification for JAX-RS.
JSR-224 Java API for XML-based Web Services 2.2	The specification for JAX-WS.
RESTEasy	The annotation-based web service implementation.
JBossWS	The SOAP service EAP implementation.

# Creating REST Services with JAX-RS

## Objectives

After completing this section, students should be able to:

- Create REST services with JAX-RS.
- Consume REST web services.
- Differentiate HTTP methods.

## RESTful Web Services with JAX-RS

As described in the previous section, JAX-RS is the Java API used to create RESTful web services. REST web services are designed to be as simple as possible to improve usability both for the developers of the services and for the clients consuming the services. In order to maintain a level of simplicity, web services must adhere to several standards in order to be considered RESTful. By implementing a web service layer, developers can abstract the front-end layer and create an application comprised of many loosely coupled components. This type of architecture is known as a client-server architecture, and it is a requirement for REST web services.

Another defining feature of RESTful web applications is that the services are stateless. Each request made to a RESTful web service must provide a response that contains all of the required information needed by the client, but the service cannot be responsible for maintaining any information regarding the session state. This restriction ensures that the client is responsible for maintaining the session state rather than needlessly complicating the REST web service.

The JEE version of the To Do List application utilizes a RESTful web service to abstract the Angular front end from the business logic and data layer. By creating specific endpoints, such as the endpoint to get a list of all of the To Do List items, clients can get the information in the JSON format to make it easier to parse. For example, the following illustrates an HTTP GET request on the To Do List application's `lookupItemById()` REST endpoint, which returns a To Do List item from the MySQL database based on the item's ID:

```
[student@workstation todojee]$ curl localhost:8080/todo/api/items/1
```

The service returns the following JSON response:

```
{"id":1,"description":"Pick up newspaper","done":false,"user":null}
```

Any client with access to service can retrieve this information, allowing for multiple front-end applications to leverage the same data and business logic for the To Do List application. Clients only need to be able to make HTTP requests and to parse the responses from the service in order to consume the REST service.

Java EE 7 supports JAX-RS 2.0, which makes developing RESTful web services and adhering to their standards very simple. JAX-RS utilizes several annotations in order to define the behavior of the web services. These annotations are placed directly in the service class to create different types of endpoints and to define parameters. To facilitate the development of web services, JBoss Developer Studio has a wizard to create all necessary files to define a RESTful web service.

## Creating RESTful Web Services

A JAX-RS RESTful web service consists of one or more classes utilizing the JAX-RS annotations to create a web service. The first step to create the web service is to create a class that extends the class `javax.ws.rs.core.Application`. In addition to declaring the RESTful web service, the new subclass is used to also define the base URI for the web service with the `@ApplicationPath` annotation. The following is an example of a class that extends `javax.ws.rs.core.Application`:

```
import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@ApplicationPath("/api")
public class Service extends Application {
 //Can be left empty
}
```

The `@ApplicationPath` annotation sets the base URI for the web service. In the previous example, the application path is configured to be `/api`. As a result, all requests to this service must be preceded by `/api` in the URI. For example, if the application's context path is `helloworld` and the application is available at `http://localhost:8080/helloworld`, then the REST service is exposed at `http://localhost:8080/helloworld/api/`. This URI is often further expanded when adding additional endpoints and paths.



### Note

Optionally, you can use the new subclass to provide custom implementations for some methods in the parent class, however it is not required to do any further modification than the given example in order to implement a RESTful web service.

An alternative to subclassing the `javax.ws.rs.core.Application` class is using the `web.xml` in the application to define the `javax.ws.rs.core.Application` class and specify the base URI. The following example `web.xml` provides the same functionality as the previous Java-based example without requiring the creation of an additional Java class:

```
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/
 xml/ns/javaee/web-app_3_0.xsd">
 <servlet>
 <servlet-name>javax.ws.rs.core.Application</servlet-name>
 </servlet>
 <servlet-mapping>
 <servlet-name>javax.ws.rs.core.Application</servlet-name>
 <url-pattern>/api/*</url-pattern>
 </servlet-mapping>
 ...
</web-app>
```

## The RESTful Root Resource Class

Using the available JAX-RS annotations is an easy way to create a RESTful web service from an existing POJO class. For example, consider a basic POJO, such as the following:

```
public class HelloWorld {

 public String hello() {
 return "Hello World!";
 }
}
```

By annotating this class and method with the following annotations, the method becomes exposed as an endpoint within the RESTful web service:

```
@Stateless
@Path("hello")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class HelloWorld {

 @GET
 public String hello() {
 return "Hello World!";
 }
}
```

This example service exposes the `hello()` method to a standard HTTP GET request. Therefore, accessing `http://localhost:8080/hello-world/api/hello` using an HTTP GET request (such as with a web browser) returns the String `Hello World!` in JSON.

Recall that the `@ApplicationPath` annotation on the `Application` class dictates the base URI for all requests to that application. Additionally, within the RESTful service class, the class and the individual methods and endpoints can be designated with their own specific `@Path` value. This allows users of the service to more intuitively access a resource.

Looking at the previous example, notice that the `@Path` annotation at the class level is set to `hello`. This annotation creates another layer in the URI, in addition to the existing `@ApplicationPath`. The following sample shows another example of modifying the `@Path` annotation:

```
@Stateless
@Path("hello")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class HelloWorld {

 @GET
 @Path("person/{id}/name")
 public String hello(String id) {
 return "Hello" + id + "!";
 }
}
```

In this instance, this method is called when making a request to the following URI: `http://localhost:8080/hello-world/api/hello/person/1/name`. The `{id}` part of the path is a variable, which is denoted by the surrounding braces, and its value must be provided by the client in the URI for every request.

The following table summarizes the available annotations for the remainder of the RESTful root resource class:

### JAX-RS Annotations

Annotation	Description
@ApplicationPath	The @ApplicationPath annotation is applied to the subclass of the javax.ws.rs.core.Application class and defines the base URI for the web service.
@Path	The @Path annotation defines the base URI for either the entire root class or for an individual method. The path can contain either an explicit static path, such as hello, or it can contain a variable to be passed in on the request. This value is referenced using the @PathParam annotation.
@Consumes	The @Consumes annotation defines the type of the request's content that is accepted by the service class or method. If an incompatible type is sent to the service, the server returns HTTP error 415, "Unsupported Media Type." Acceptable parameters include application/json, application/xml, text/html, or any other MIME type.
@Produces	The @Produces annotation defines the type of the response's content that is returned by the service class or method. Acceptable parameters include application/json, application/xml, text/html, or any other MIME type.
@GET	The @GET annotation is applied to a method to create an endpoint for the HTTP GET request type, commonly used to retrieve data.
@POST	The @POST annotation is applied to a method to create an endpoint for the HTTP POST request type, commonly used to save or create data.
@DELETE	The @DELETE annotation is applied to a method to create an endpoint for the HTTP DELETE request type, commonly used to delete data.
@PUT	The @PUT annotation is applied to a method to create an endpoint for the HTTP PUT request type, commonly used to update existing data.
@PathParam	The @PathParam annotation is used to retrieve a parameter passed in through the URI, such as <code>http://localhost:8080/hello-web/api/hello/1</code> .
@QueryParam	The @QueryParam annotation is used to retrieve a parameter passed in through the URI as a query parameter, such as <code>http://localhost:8080/hello-web/api/hello?id=1</code> .

## Customizing Requests and Responses

One of the strengths of JAX-RS is the ability to customize both the MIME type of the request and response. For organizations that require enforcing a specific type of request or response, such as XML, the @Produces or @Consumes annotations can be modified to enforce XML as the

response and request type, respectively. You may prefer to use JSON, as it is a lightweight MIME type compared to XML and may reduce bandwidth.

The annotation `@Produces` defines the MIME media type for the response returned by the service. The annotation `@Consumes` defines the MIME media type for the request required by the service. Both `@Produces` and `@Consumes` can be applied at either the method level, the class level, or both. If applied to both, the annotation at the method level takes precedence and overrides the class-level annotation. If no annotation for either `@Produces` or `@Consumes` is defined on the method, the method defaults to the MIME type defined at the class level.

The following example defines a JAX-RS class that demonstrates the use of the `@Produces` and `@Consumes` annotation:

```
@Stateless
@Path("hello")
@Produces(MediaType.APPLICATION_JSON)
public class HelloWorld {

 @GET
 @Produces("text/html")
 public String hello() ①{
 return "Hello World!";
 }

 @GET
 @Path("newest")
 public Person getNewestPerson() ②{
 ...implementation omitted...
 }

 @POST
 @Consumes(MediaType.APPLICATION_JSON)
 public String savePerson(Person person) ③ {
 ...implementation omitted...
 }
}
```

- ① The `hello()` method returns output that the client must expect to be in HTML. This is dictated by the method level `@Produces("text/html")` annotation.
- ② The `getNewestPerson()` method returns output that the client must expect to be in JSON. This is dictated by the class level `@Produces(MediaType.APPLICATION_JSON)` because there is no method level `@Produces` annotation.
- ③ The `savePerson(Person person)` method requires that the request is in JSON, otherwise the client receives an HTTP 415 error for unsupported media type.

## HTTP Methods

The HTTP protocol defines several methods through which the protocol enacts different actions. The client is responsible for specifying the type of request in addition to the path of the request. The following is a list of the methods:

- **GET**: The GET method retrieves data.
- **POST**: The POST method creates a new entity.

- **DELETE:** The DELETE method removes an entity.
- **PUT:** The PUT method updates an entity.

Each HTTP method has a similarly named annotation that is used to annotate methods in a RESTful service class. If two Java methods exist on the same path, JAX-RS determines which method to use by matching the HTTP method on the HTTP request made by the client and the annotation on the method. The following is an example of a RESTful web service class:

```

@Stateless
@Path("hello")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class HelloWorld {

 @GET①
 @Path("person")
 public List<Person> getPersons() {
 ...implementation omitted...
 }

 @POST②
 @Path("person")
 public String savePerson(Person person) {
 ...implementation omitted...
 }

 @PUT③
 @Path("person")
 public String updatePerson(Person person) {
 ...implementation omitted...
 }

 @DELETE④
 @Path("person/{id}")
 public String deletePerson(@PathParam("id") String id) {
 ...implementation omitted...
 }
}

```

- ① This method returns a JSON representation of the Java list of Person objects when an HTTP GET request is made to the following URL: <http://localhost:8080/hello-world/hello/person>.
- ② This method creates a Person object when an HTTP POST request with the JSON representation of a Person is made to the following URL: <http://localhost:8080/hello-world/hello/person>.
- ③ This method updates a Person object when an HTTP PUT request with the JSON representation of an existing Person is made to the following URL: <http://localhost:8080/hello-world/hello/person>.
- ④ This method deletes a Person object when an HTTP DELETE request is made to the following URL: <http://localhost:8080/hello-world/hello/person/1>.

Notice that the GET, POST, and PUT methods are all on the same path, however the client is able to dictate which endpoint is reached based on the HTTP method that is specified on the request.

## Injecting Parameters from the URI

In many instances, clients using RESTful web services need to request specific information from a service. This is accomplished either by providing parameters in the URI, either as a path parameter or a query parameter.

To use a path parameter on a JAX-RS method, annotate it with the `@PathParam` annotation. This annotation is typically used when the client is requesting a specific resource, such as requesting a user's data. The following is an example of a path parameter:

```
@GET
@Path("{id}")①
public Person getPerson(@PathParam("id")② Long id) {
 return entityManager.find(Person.class, id);
}
```

- ① The variable `id` is passed in by the client in the URI as part of the path. For example, the following request is compatible with this method: `http://localhost:8080/hello-web/api/hello/1`. The `1` is passed into the `getPerson()` method.
- ② The `@PathParam` annotation maps the variable from the path to the Java method parameter.

To use a query parameter on a JAX-RS method, annotate it with the `@QueryParam` annotation. This annotation is typically used in searches and when filtering data, such as filtering users by their email preferences. The following is an example of a query parameter used to determine which users want emails sent to them based on the variable `sendEmail`:

```
@GET
@Path("users")
public List<Person> getUsers(@QueryParam("sendEmail") String sendEmail①) {
 return entityManager.find("SELECT * USERS WHERE user.sendEmail=" + sendEmail);②
}
```

- ① The `@QueryParam` annotation maps the value that is used in the URI for `sendEmail` to a Java String of the same name. The following is an example URI that passes in a `false` flag to the `sendEmail` variable: `http://localhost:8080/hello-world/api/users?sendEmail=false`.
- ② The `return` method leverages the mapped value to the Java variable by using it to query the database and filter all of the users by the parameter. In this instance, the query parameter lists the users based on the value of the `sendEmail` variable.

In many instances, a query parameter requires a default value to both prevent the client's request from failing for not providing a parameter and to allow the developers of the service to create a single method for searching and filtering without making all parameters required. The following is the same example with the default value for the `sendEmail` variable set to `False`:

```
@GET
@Path("users")
public List<Person> getUsers(@DefaultValue("True") @QueryParam("sendEmail") String
sendEmail) {
 return entityManager.find("SELECT * FROM USERS WHERE user.sendEmail=" + sendEmail);
}
```

With the default value set to **True**, the client is no longer required to provide the value in the URI and can instead make a request to `http://localhost:8080/hello-world/api/users` to receive a list of users that are accepting emails.



## References

Further information is available in the *Developing Web Services Guide* for Red Hat JBoss EAP 7; at [https://access.redhat.com/documentation/en-us/red\\_hat\\_jboss\\_enterprise\\_application\\_platform/](https://access.redhat.com/documentation/en-us/red_hat_jboss_enterprise_application_platform/)

## ► Guided Exercise

# Exposing a REST Service

In this exercise, you will expose a REST API for an application.

## Outcomes

You should be able to create a RESTful web service and test the service using **curl**.

## Before You Begin

Open a terminal window on the **workstation** VM and run the following command to download the lab files that are required for this workshop.

```
[student@workstation ~]$ lab hello-rest setup
```

## Steps

- 1. Import the hello-rest project into JBoss Developer Studio IDE (JBDS).
  - 1.1. Double-click the JBDS icon on the **workstation** VM desktop to start.
  - 1.2. Enter **/home/student/JB183/workspace** in the **Workspace** field in the **Eclipse Launcher** window, and then click **Launch**.
  - 1.3. In the JBDS menu, click **File > Import** to open the **Import** wizard.
  - 1.4. On the **Select** page, click **Maven > Existing Maven Projects**, and then click **Next**.
  - 1.5. In the **Maven projects** page, click **Browse** to open the **Select root folder** window. Navigate to the **/home/student/JB183/labs/** directory. Select the **hello-rest** folder, and then click **OK**.
  - 1.6. On the **Maven projects** page, click **Finish**.
  - 1.7. Watch the JBDS status bar to monitor the progress of the import operation. It may take a few minutes to download all required dependencies.
- 2. Create the root context for the web service.
  - 2.1. In the expanded **hello-rest** item in the **Project Explorer** tab in the left pane of JBDS, select **hello-rest > Java Resources > src/main/java > com.redhat.training.rest** and expand the package.
  - 2.2. Right-click **com.redhat.training.rest** and click **New > Class**.
  - 2.3. In the **Name** field, enter **Service**. Click **Finish**.
  - 2.4. In the new class, add the **@ApplicationPath** annotation, import the library, and specify the path as **/api**:

```
package com.redhat.training.rest;

import javax.ws.rs.ApplicationPath;

@ApplicationPath("/api")
public class Service {
}
```

- 2.5. Complete the class by importing and extending the `javax.ws.rs.core.Application` class:

```
package com.redhat.training.rest;

import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@ApplicationPath("/api")
public class Service extends Application {
}
```

- 2.6. Save your changes by pressing **Ctrl+S**.

- 3. Open and update the `PersonService.java` RESTful web service in the `com.redhat.training.rest` package to be stateless using the `@Stateless` annotation.

```
//TODO Add the stateless annotation
@Stateless
```

- 4. Add the `@Path` annotation to make the end points for this web service class available at `http://localhost:8080/hello-rest/api/persons/`:

```
//TODO Add a Path for persons
@Path("persons")
```

- 5. Define the `@Consumes` and `@Produces` media type for the service.

- 5.1. Add the `@Consumes` annotation to allow the service to consume JSON:

```
//TODO Add a Consumes annotation for JSON
@Consumes(MediaType.APPLICATION_JSON)
```

- 5.2. Add the `@Produces` annotation to allow the service to produce JSON:

```
//TODO Add a Produces annotation for JSON
@Produces(MediaType.APPLICATION_JSON)
```

- 6. Configure the `getPerson()`, `getAllPersons()`, `deletePerson()`, and `savePerson()` methods in the `PersonService.java` class to be available as REST endpoints.

- 6.1. Expose the `getPerson(Long id)` method by adding the `@GET` annotation:

```
//TODO add GET annotation
@GET
public Person getPerson(Long id) {
 return entityManager.find(Person.class, id);
}
```

- 6.2. Update the `getPerson(Long id)` method to allow consumers of the REST service to request a Person with a specific ID using the REST endpoint by adding the `@Path` and `@PathParam` annotations:

```
@GET
//TODO add path for ID
@Path("{id}")
public Person getPerson(@PathParam("id") Long id) {
 return entityManager.find(Person.class, id);
}
```

A GET request to `http://localhost:8080/hello-rest/api/persons/3` now returns the JSON representation of the Person with ID 3.

- 6.3. Add a `@GET` annotation to the `getAllPersons()` method to expose the method as a REST endpoint:

```
//TODO add GET annotation
@GET
public List<Person> getAllPersons() {
 TypedQuery<Person> query = entityManager.createQuery("SELECT p FROM Person p",
 Person.class);
 List<Person> persons = query.getResultList();

 return persons;
}
```

A GET request to `http://localhost:8080/hello-rest/api/persons/` now returns the JSON representation of all Person objects in the database.

- 6.4. Add the annotation for `@DELETE` to the `deletePerson(Long id)` method to allow HTTP DELETE requests to remove a Person from the database:

```
//TODO add DELETE annotation
@DELETE
public void deletePerson(Long id) {
try {
try {
tx.begin();
entityManager.remove(getPerson(id));
} finally {
tx.commit();
}
} catch (Exception e) {
throw new EJBException();
}
}
}
```

- 6.5. Similar to the method that returns an individual Person, the `deletePerson` method requires an ID parameter to remove a specific Person from the database. Update the method with a `@Path` and a `PathParam` annotation to allow users to pass in that parameter in the HTTP request:

```
@DELETE
//TODO add Path for ID
@Path("{id}")
public void deletePerson(@PathParam("id") Long id) {
try {
try {
tx.begin();
entityManager.remove(getPerson(id));
} finally {
tx.commit();
}
} catch (Exception e) {
throw new EJBException();
}
}
}
```

A DELETE request to `http://localhost:8080/hello-rest/api/persons/3` now removes the Person with ID 3 from the database.

- 6.6. Add a `@POST` annotation to the `savePerson(Person person)` method to create an endpoint for saving a Person object to the database:

```
//TODO add POST annotation
@POST
public Response savePerson(Person person) {
try {
...
}
}
```

A POST request to `http://localhost:8080/hello-rest/api/persons/` with a JSON representation of a Person now persists that person to the database.

- 6.7. Save your changes by pressing **Ctrl+S**.
- 7. Start EAP by selecting the **Servers** tab in the bottom pane of JBDS. Right-click the server **Red Hat JBoss EAP 7.0 [Stopped]** and click on the green start button to start the server.
- 8. Deploy the hello-rest application using the following commands in the terminal window:

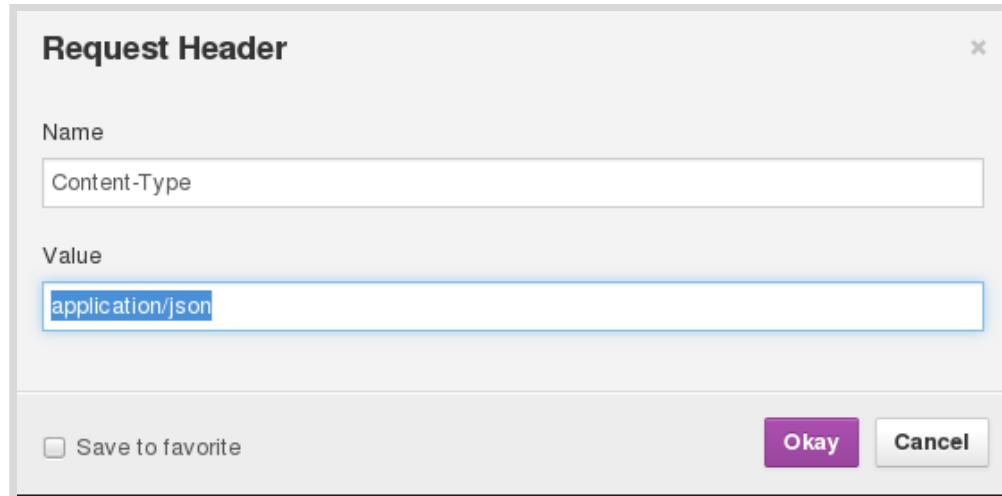
```
[student@workstation ~]$ cd /home/student/JB183/labs/hello-rest
[student@workstation hello-rest]$ mvn wildfly:deploy
```

- 9. Test the REST API using **curl** and the Firefox REST Client plug-in.
- 9.1. Start Firefox on the **workstation** VM and click the REST Client plug-in in the browser's toolbar.



**Figure 6.2: The Firefox REST client plug-in**

- 9.2. In the top toolbar, click **Headers**, and select **Custom Header** to add a new custom header to the request.
- 9.3. In the custom header dialog, enter the following information:
- Name: **Content-Type**
  - Value: **application/json**



**Figure 6.3: Creating a custom request header in the REST Client**

- Click **Okay**.
- 9.4. Select **POST** as the **Method**. In the URL form, enter **http://localhost:8080/hello-rest/api/persons**.
- 9.5. In the **Body** section of the request, add the following JSON representation of a **Person** entity:

```
{"name": "Shadowman"}
```

Click Send.

- 9.6. Verify in the **Response Headers** tab that the **Status Code** is **200 OK**.

Response Headers	Response Body (Raw)	Response Body (Highlight)
<pre>1. Status Code      : 200 OK 2. Connection       : keep-alive 3. Content-Length   : 0 4. Date             : Fri, 20 Oct 2017 20:27:00 GMT 5. Server            : JBoss-EAP/7 6. X-Powered-By     : Undertow/1</pre>		

Figure 6.4: Creating a custom request header in the REST Client

- 9.7. Change the **Body** of the request to the following to trigger the bean validation for the **Person** entity that requires that the **name** attribute has more than two characters:

```
{"name": "a"}
```

Click **Send** and observe that the response returns with a status code of **500** to indicate a server error.

- 9.8. In a terminal window, use the following **curl** command in the terminal window to make an HTTP GET request to retrieve all **Persons**:

```
[student@workstation hello-rest]$ curl \
http://localhost:8080/hello-rest/api/persons
```

Look for the **Person** created in the previous step in the output:

```
[{"id":1, "name": "Shadowman"}]
```

- 9.9. Delete the newly created **Person** using the following **DELETE** HTTP request with the correct **Person** ID:

```
[student@workstation hello-rest]$ curl -X DELETE \
http://localhost:8080/hello-rest/api/persons/1
```

- 9.10. Run the following command to verify that the **Person** was deleted:

```
[student@workstation hello-rest]$ curl http://localhost:8080/hello-rest/api/
persons
```

The output displays that the Person is no longer in the database.

► **10.** Undeploy the application and stop EAP.

- 10.1. In the terminal window, run the following command to undeploy the application from EAP:

```
[student@workstation hello-rest]$ mvn wildfly:undeploy
```

- 10.2. Right click on the `hello-rest` project in the **Project Explorer**, and select **Close Project** to close this project.
- 10.3. Right-click **Red Hat JBoss EAP 7.0** in the **Servers** tab and then click **Stop** to stop the EAP instance.

This concludes the guided exercise.

# Consuming a REST Service

## Objectives

After completing this section, students should be able to:

- Invoke a REST API remotely.
- Describe the HTTP status codes.

## Consuming REST Services

One of the primary advantages of REST services is that there are many ways to interact with or consume the services. Developers are not limited to a specific language or a specific approach to utilize a REST service. As seen in the previous exercise, you can use a simple `curl` command to reach a REST endpoint or use a browser plugin to test a REST service.

While `curl` and other basic HTTP request approaches offer a simpler and faster approach for interacting with and testing a REST service, many developers prefer to use language-specific libraries to programmatically interface with REST services to provide greater functionality and control of the requests and responses.

In Java, for example, JAX-RS 2.0 provides a new API that can send and receive HTTP requests to local and remote RESTful web services. There are three distinct components of a JAX-RS client:

- **Client:** The client creates instances of `WebTargets`.
- **WebTarget:** The URL destination of a REST endpoint.
- **Response:** The object containing the response from the service.

The first step in creating a REST client is to initialize a `Client` object using the `ClientBuilder`. The following code creates a new `Client`:

```
Client client = ClientBuilder.newClient();
```

At the end of the request, be sure to also close the `Client` using the following code:

```
client.close();
```

Next, use the newly created `Client` to create a `WebTarget` instance based on the REST endpoint. The following code creates a `WebTarget` that connects to a locally-running REST service, available at `http://localhost:8080/todo/api/users`:

```
Client client = ClientBuilder.newClient();
WebTarget webTarget = client.target("http://localhost:8080/todo/api/users");
```

Finally, to get the `Response` from the web service, use the `webTarget` to configure and build the request. The following example builds and executes an HTTP GET request to the `webTarget`'s REST endpoint and returns a `Response` object:

```
Client client = ClientBuilder.newClient();
WebTarget webTarget = client.target("http://localhost:8080/todo/api/users");
Response response = webTarget.request().get();
```

## Configuring the Target

Many developers find that a client needs multiple WebTargets to interact with different REST endpoints to fulfill the requirements of a request or, more commonly, developers need to parameterize the request. To do this, developers can append additional elements to extend the WebTarget's path. The following example demonstrates creating an additional path WebTarget instance that targets `http://localhost:8080/todo/api/items`:

```
WebTarget webTarget = client.target("http://localhost:8080/todo/api");
WebTarget items = webTarget.path("items");
```

The `path()` method can also create a parameterized URL when a REST service requires a path parameter. For example, the To Do List application's REST service requires an ID when trying to return a single to-do item. The following is the `curl` command syntax to return the item based on its ID:

```
[student@workstation ~]$ curl http://localhost:8080/todo/api/items/{id}
```

And the following example demonstrates requesting the item with an ID of 1:

```
[student@workstation ~]$ curl http://localhost:8080/todo/api/items/1
```

Using the JAX-RS programmatic client is very similar to using `curl`. The WebTarget appends the path variable and then uses the `resolveTemplate` method to replace the variable with the desired value. The following example creates the same request as the previous `curl` command:

```
WebTarget webTarget = client.target("http://localhost:8080/todo/api");
WebTarget itemTarget = webTarget.path("items/{id}").resolveTemplate("id", "1");
```

In addition to supporting path variables, the JAX-RS client library also supports query parameters. The following syntax generates a request to the URL `http://localhost:8080/todo/api/items?id=1`:

```
WebTarget webTarget = client.target("http://localhost:8080/todo/api");
WebTarget itemTarget = webTarget.path("items").QueryParam("id", "1");
```

## Creating the Request

The `request()` method on the WebTarget class enables developers to define the type of HTTP request to make to the REST endpoint. For example, the following request represents a GET request on the WebTarget and returns a response object:

```
Response response = webTarget.request().get();
```

A POST request uses the `post()` method to send data in a specific format, such as XML or JSON, to the REST service. For example, the following code makes a POST request to a `WebTarget`, sending an `Item` entity formatted as JSON:

```
Item item = new Item(1, "Clean the car", "false")
Response response = webTarget.request().post(Entity.json(item));
```

Each HTTP request type has a corresponding method. The following methods are available for HTTP requests:

- `get()`
- `post()`
- `delete()`
- `put()`

## Parsing the Response

After making a request with the `WebTarget`, the target returns a `Response` object. This `Response` object needs to be mapped to the expected entity or object type. For example, if the request is for a single field, the response can be mapped to a `String`. If the request is for a complex object, like an `Item` from the To Do List application, the `Item` needs to be mapped to `Item.class`. The following demonstrates a full client request to get a specific `Item` with an ID of 1 that maps the `Response` object to an `Item` object:

```
Client client = ClientBuilder.newBuilder().build(); ①
WebTarget webTarget = client.target("http://localhost:8080/todo/api"); ②
WebTarget itemTarget = webTarget.path("items/{id}").resolveTemplate("id", "1"); ③
Response response = itemTarget.request().get(); ④
Item foundItem = response.readEntity(Item.class); ⑤
client.close(); ⑥
```

- ① Create a new `Client` instance using `ClientBuilder`.
- ② Define the base for a new `WebTarget` instance.
- ③ Create another `WebTarget` instance that builds on the path of the base `WebTarget`, requesting the `Item` with ID 1.
- ④ Use the `WebTarget` to make a GET request.
- ⑤ Map the `Response` object to an `Item` entity.
- ⑥ Close the client when there are no more `WebTargets` to initialize.

## Authentication with REST Clients

In many instances, REST APIs are secured with some kind of security, such as Basic or Digest authentication. Basic authentication is often the simplest and most common form of restricting access to REST APIs. This requires developers to provide a credentials encrypted with a Base 64 Encoder when making requests to the REST API. The following is an example request for a resource secured with Basic authentication:

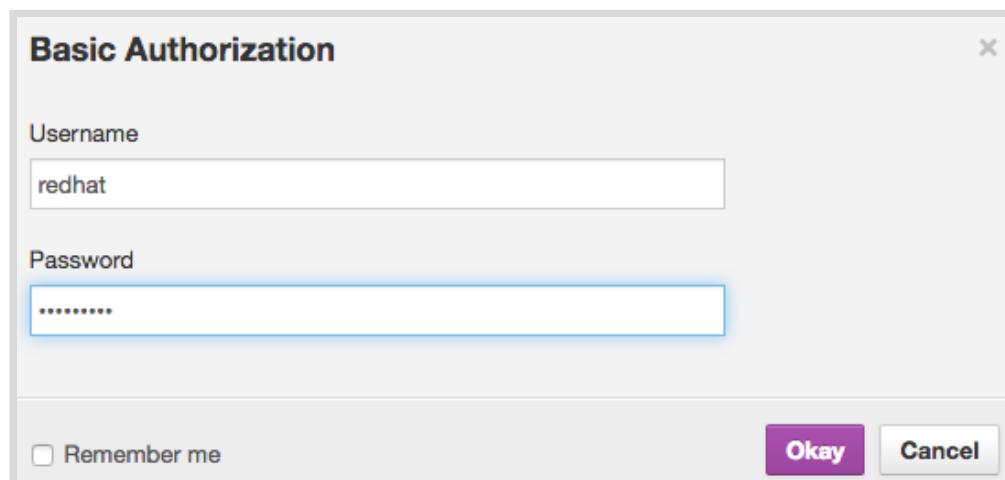
```

String username = "redhat"; ①
String password = "Shadowman"; ②
String userpass = username + ":" + password; ③
String encodedCred = new BASE64Encoder().encode(userpass.getBytes()); ④
Client client = ClientBuilder.newBuilder().build();
WebTarget webTarget = client.target("http://localhost:8080/todo/api");
WebTarget itemTarget = webTarget.path("items/{id}").resolveTemplate("id", "1");
Response response = itemTarget.request().header("Authorization", "Basic " +
 encodedCred).get(); ⑤
Item foundItem = response.readEntity(Item.class);
client.close();

```

- ① A string of the user name that is able to access the requested resource.
- ② The password for the user name.
- ③ The string representing the required format for the Basic authentication header. The syntax is *username:password*.
- ④ The string containing the Base 64 encoded credentials.
- ⑤ The request requires an "Authorization" header with the payload containing a string in the following format: "Basic <encodedCredentials>".

For testing purposes, the REST Client plug-in in Firefox provides developers the ability to add Authentication headers for accessing resources protected with Basic authentication. To add authentication, click **AuthenticationBasic Authentication**.



**Figure 6.5: The Basic authentication credentials dialog**

Enter the user credentials and click **Okay**. The **Headers** section contains the automatically-encoded credentials for the HTTP request.

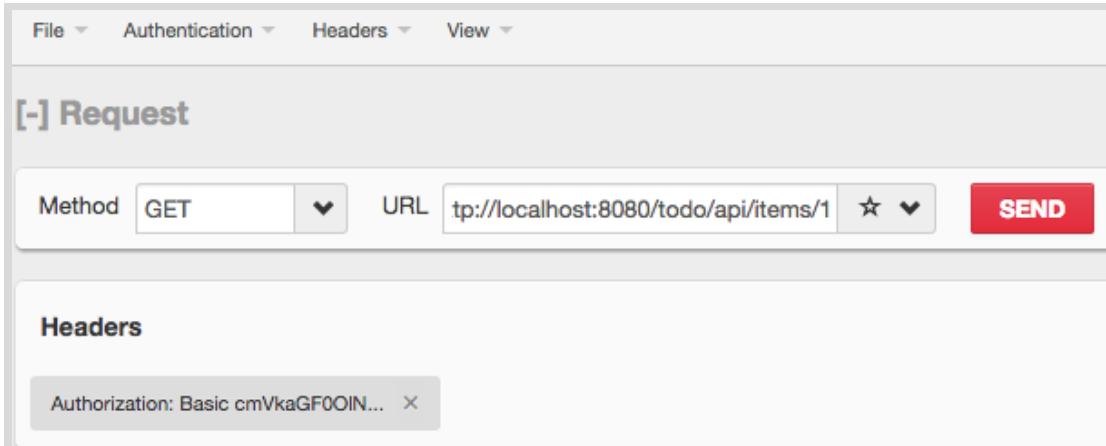


Figure 6.6: The REST client authentication header

## HTTP Status Codes

When making an HTTP request, the server returns a status code to tell the client whether the request was successful or if an error occurred, indicating which type of error using a variety of numeric codes. The most common HTTP status codes:

### HTTP Status Code Table

HTTP Status Code	Description
200	"OK." The request is successful.
400	"Bad Request." The request is malformed or it is pointing to the wrong endpoint.
403	"Forbidden." That client did not provide the correct credentials.
404	"Not Found." The path or endpoint is not found or the resource does not exist.
405	"Method Not Allowed." The client attempted to use an HTTP method on an endpoint that does not support it.
409	"Conflict." The requested object cannot be created because it already exists.
500	"Internal Server Error." The server failed to process the request. Contact the owner of the REST service to investigate the reason.



### References

#### JSR-311 Java API for RESTful Web Services 2.0

<http://www.jcp.org/en/jsr/detail?id=311>

#### JSR-224 Java API for XML-based Web Services 2.2

<http://www.jcp.org/en/jsr/detail?id=224>



## References

Further information is available in the *Developing Web Services Guide for Red Hat JBoss EAP 7* at

[https://access.redhat.com/documentation/en-us/  
red\\_hat\\_jboss\\_enterprise\\_application\\_platform/](https://access.redhat.com/documentation/en-us/red_hat_jboss_enterprise_application_platform/)

## ► Quiz

# Consuming a REST Service

Match the items below to their counterparts in the table.

"Bad Request."

"Internal Server Error."

"OK."

An object returned from a request.

Creates instances of objects pointing to REST endpoints.

Represents a REST endpoint.

Term	Definition
HTTP Status Code 200	
HTTP Status Code 400	
HTTP Status Code 500	
WebTarget	
Client	
Response	

## ► Solution

# Consuming a REST Service

Match the items below to their counterparts in the table.

Term	Definition
HTTP Status Code 200	"OK."
HTTP Status Code 400	"Bad Request."
HTTP Status Code 500	"Internal Server Error."
WebTarget	Represents a REST endpoint.
Client	Creates instances of objects pointing to REST endpoints.
Response	An object returned from a request.

## ► Lab

# Creating REST Services

In this lab, you will learn how to expose the To Do List Java EE application's REST Service.

## Outcomes

You should be able to expose and test the To Do List Java EE application's REST service.

## Before You Begin

Open a terminal window on the `workstation` VM and run the following command to download the files required for this lab.

```
[student@workstation ~]$ lab todo-rest setup
```

## Steps

1. Import the `todo-rest` project into JBoss Developer Studio IDE (JBDS).
2. Create the root context class named `JaxRsActivator.java` in the `com.redhat.training.todo.rest` package for the REST service. Set the `ApplicationPath` to `/api`.
3. Create and configure the `Path` annotation for the `com.redhat.training.todo.rest.ItemResourceRESTService.java` class to make the endpoints for this web service class available at `http://localhost:8080/todo-rest/api/items/`.
4. There are four methods in the `ItemResourceRESTService.java` class that need to be exposed as a REST endpoint:
  - `listAllItems()`
  - `lookupItemById(long id)`
  - `createItem(Item item)`
  - `deleteItem(long id)`Update the `listAllItems()` method in the `ItemResourceRESTService.java` class to meet the following requirements:
  - Available with a GET request.
  - Produces JSON.
5. Update the `lookupItemById(long id)` method in the `ItemResourceRESTService.java` class to meet the following requirements:
  - Available with a GET request.
  - Produces JSON.

- Create a path parameter for the `id` so that an individual `item` is available with the following URL: `http://localhost:8080/todo-rest/api-items/{id}`
6. Update the `createItem(Item item)` method in the `ItemResourceRESTService.java` class to meet the following requirements:
- Available with a POST request.
  - Produces and consumes JSON.
7. Update the `deleteItem(long id)` method in the `ItemResourceRESTService.java` class to meet the following requirements:
- Available with a DELETE request.
  - Create a path parameter for the `id` so that an individual `item` is deleted with DELETE request to the following URL: `http://localhost:8080/todo-rest/api-items/{id}`
8. Start EAP by selecting the **Servers** tab in the bottom pane of JBDS. Right-click the server **Red Hat JBoss EAP 7.0 [Stopped]** and click on the green start button to start the server.
9. Deploy the todo-rest application using the following commands in the terminal window:

```
[student@workstation ~]$ cd /home/student/JB183/labs/todo-rest
[student@workstation todo-rest]$ mvn wildfly:deploy
```

10. Create an `Item` using the Firefox REST client plug-in by making a POST request to `http://localhost:8080/todo-rest/api/items`
11. Test the functionality for listing all items and delete the `Item` added in the previous step using either `curl` or the Firefox plug-in.
12. Cleanup and grade.

- 12.1. To verify that you have successfully deployed the application, open a new terminal window on your `workstation` VM and run the following command to grade this lab:

```
[student@workstation ~]$ lab todo-rest grade
```

If you see failures after running the command, look at the errors, troubleshoot the deployment, and fix the errors. Rerun the grading script and verify that it passes.

- 12.2. In the terminal window where you ran the Maven command to deploy the application, run the following command to undeploy the application from EAP:

```
[student@workstation todo-rest]$ mvn wildfly:undeploy
```

- 12.3. Right-click on the `todo-rest` project in the **Project Explorer**, and select **Close Project** to close this project.
- 12.4. Right-click **Red Hat JBoss EAP 7.0** in the **Servers** tab, and then click **Stop** to stop the EAP instance.

This concludes the lab.

## ► Solution

# Creating REST Services

In this lab, you will learn how to expose the To Do List Java EE application's REST Service.

### Outcomes

You should be able to expose and test the To Do List Java EE application's REST service.

### Before You Begin

Open a terminal window on the **workstation** VM and run the following command to download the files required for this lab.

```
[student@workstation ~]$ lab todo-rest setup
```

### Steps

1. Import the todo-rest project into JBoss Developer Studio IDE (JBDS).
  - 1.1. Click the JBDS icon on the **workstation** VM desktop to start JBDS.
  - 1.2. In the JBDS menu, click **File > Import** to open the **Import** wizard.
  - 1.3. On the **Select** page, select **Maven > Existing Maven Projects**, and then click **Next**.
  - 1.4. In the **Maven projects** page, click **Browse** to open the **Select root folder** window.
  - 1.5. Navigate to the **/home/student/JB138/labs/** directory. Select the **todo-rest** folder, and then click **OK**.
  - 1.6. On the **Maven projects** page, click **Finish**.
  - 1.7. Wait until JBDS finishes importing the project.
2. Create the root context class named **JaxRsActivator.java** in the **com.redhat.training.todo.rest** package for the REST service. Set the **ApplicationPath** to **/api**.
  - 2.1. In the expanded **todo-rest** item in the **Project Explorer** tab in the left pane of JBDS, select **todo-rest > Java Resources > src/main/java > com.redhat.training.todo.rest** and expand the package.
  - 2.2. Right-click **com.redhat.training.todo.rest** and click **New > Class**.
  - 2.3. In the **Name** field, enter **JaxRsActivator**. Click **Finish**.
  - 2.4. In the new class, add the **@ApplicationPath** annotation, import the library, and specify the path as **/api**:

```
package com.redhat.training.todo.rest;

import javax.ws.rs.ApplicationPath;

@Path("/api")
public class JaxRsActivator {

}
```

- 2.5. Complete the class by importing and extending the `javax.ws.rs.core.Application` class:

```
package com.redhat.training.todo.rest;

import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@Path("/api")
public class JaxRsActivator extends Application {

}
```

- 2.6. Save your changes by pressing **Ctrl+S**.

3. Create and configure the `Path` annotation for the `com.redhat.training.todo.ItemResourceRESTService.java` class to make the endpoints for this web service class available at `http://localhost:8080/todo-rest/api/items/`.

Open and update the `ItemResourceRESTService.java` RESTful web service in the `com.redhat.training.todo` package to set the `@Path` for the service to `/items`:

```
// *Add the path annotation*
@Path("/items")
@RequestScoped
public class ItemResourceRESTService {
```

4. There are four methods in the `ItemResourceRESTService.java` class that need to be exposed as a REST endpoint:

- `listAllItems()`
- `lookupItemById(long id)`
- `createItem(Item item)`
- `deleteItem(long id)`

Update the `listAllItems()` method in the `ItemResourceRESTService.java` class to meet the following requirements:

- Available with a GET request.
- Produces JSON.

- 4.1. In the `ItemResourceRESTService.java` class, add a `@GET` annotation to the `listAllItems()` method:

```
// *Add listAllItems() annotations* //
@GET
public List<Item> listAllItems() {
 return repository.findAllItemsForUser(currentUser);
}
```

- 4.2. Set the `listAllItems()` method to produce JSON with the following annotation:

```
// *Add listAllItems() annotations* //
@GET
@Produces(MediaType.APPLICATION_JSON)
public List<Item> listAllItems() {
 return repository.findAllItemsForUser(currentUser);
}
```

- 4.3. Save your changes by pressing `Ctrl+S`.

5. Update the `lookupItemById(long id)` method in the `ItemResourceRESTService.java` class to meet the following requirements:

- Available with a GET request.
- Produces JSON.
- Create a path parameter for the `id` so that an individual `item` is available with the following URL: `http://localhost:8080/todo-rest/api-items/{id}`

- 5.1. In the `ItemResourceRESTService.java` class, add a `@GET` annotation to the `lookupItemById(long id)` method:

```
// *Add lookupItemById() annotations* //
@GET
public Item lookupItemById(long id) {
```

- 5.2. Create a `@Path` annotation and set it to be the `id` parameter by configuring the `@PathParam` annotation in the method header:

```
// *Add lookupItemById() annotations* //
@GET
@Path("/{id}")
public Item lookupItemById(@PathParam("id") long id) {
```

- 5.3. Set the `lookupItemById(long id)` method to produce JSON with the following annotation:

```
// *Add lookupItemById() annotations* //
@GET
@Path("/{id}")
@Produces(MediaType.APPLICATION_JSON)
public Item lookupItemById(@PathParam("id") long id) {
```

- 5.4. Save your changes by pressing **Ctrl+S**.
6. Update the `createItem(Item item)` method in the `ItemResourceRESTService.java` class to meet the following requirements:
- Available with a POST request.
  - Produces and consumes JSON.
- 6.1. In the `ItemResourceRESTService.java` class, add a `@POST` annotation to the `createItem(Item item)` method:
- ```
// *Add createItem() annotations* //
@POST
public Response createItem(Item item) {
```
- 6.2. Set the `createItem(Item item)` method to consume JSON with the following annotation:
- ```
// *Add createItem() annotations* //
@POST
@Consumes(MediaType.APPLICATION_JSON)
public Response createItem(Item item) {
```
- 6.3. Set the `createItem(Item item)` method to produce JSON with the following annotation:
- ```
// *Add createItem() annotations* //
@POST
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response createItem(Item item) {
```
- 6.4. Save your changes by pressing **Ctrl+S**.
7. Update the `deleteItem(long id)` method in the `ItemResourceRESTService.java` class to meet the following requirements:
- Available with a DELETE request.
 - Create a path parameter for the `id` so that an individual `item` is deleted with DELETE request to the following URL: `http://localhost:8080/todo-rest/api-items/{id}`
- 7.1. In the `ItemResourceRESTService.java` class, add a `@DELETE` annotation to the `deleteItem(long id)` method:
- ```
/*Add deleteItem() annotations*/
@DELETE
public void deleteItem(long id) {
 itemService.remove(id);
}
```
- 7.2. Create a `@Path` annotation and set it to be the `id` parameter by configuring the `@PathParam` annotation in the method header:

```
//*Add deleteItem() annotations*/
@DELETE
@Path("/{id}")
public void deleteItem(@PathParam("id") long id) {
 itemService.remove(id);
}
```

7.3. Save your changes by pressing **Ctrl+S**.

8. Start EAP by selecting the **Servers** tab in the bottom pane of JBDS. Right-click the server **Red Hat JBoss EAP 7.0 [Stopped]** and click on the green start button to start the server.
9. Deploy the todo-rest application using the following commands in the terminal window:

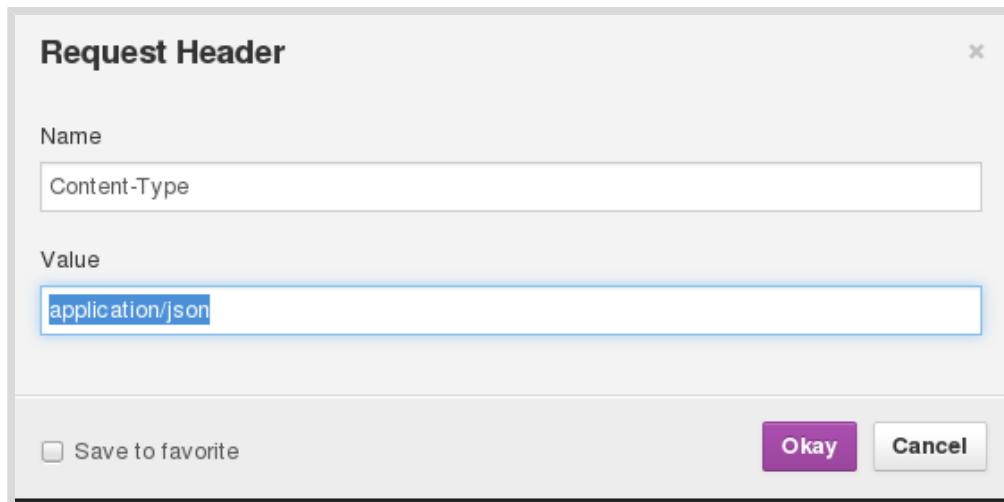
```
[student@workstation ~]$ cd /home/student/JB183/labs/todo-rest
[student@workstation todo-rest]$ mvn wildfly:deploy
```

10. Create an **Item** using the Firefox REST client plug-in by making a POST request to `http://localhost:8080/todo-rest/api/items`
- 10.1. Start Firefox on the workstation VM and click the REST Client plug-in in the browser's toolbar.



**Figure 6.8: The Firefox REST client plug-in**

- 10.2. In the top toolbar, click **Headers** and click **Custom Header** to add a new custom header to the request.
- 10.3. In the custom header dialog, enter the following information:
  - Name: **Content-Type**
  - Value: **application/json**



**Figure 6.9: Creating a custom request header in the REST Client**

Click Okay

- 10.4. Select POST as the **Method**. In the URL form, enter `http://localhost:8080/todo-rest/api/items`.
- 10.5. In the **Body** section of the request, add the following JSON representation of an **Item** entity:

```
{"description":"Complete chapter 6 lab"}
```

Click **Send**.

- 10.6. Verify in the **Response Headers** tab that the **Status Code** is 200 OK.

Response Headers	Response Body (Raw)	Response Body (Highlight)
<pre> 1. Status Code      : 200 OK 2. Connection       : keep-alive 3. Content-Length   : 0 4. Date             : Fri, 20 Oct 2017 20:27:00 GMT 5. Server            : JBoss-EAP/7 6. X-Powered-By     : Undertow/1 </pre>		

Figure 6.10: Creating a custom request header in the REST Client

11. Test the functionality for listing all items and delete the **Item** added in the previous step using either `curl` or the Firefox plug-in.
  - 11.1. In a terminal window, use the following `curl` command to make an HTTP GET request to retrieve all **Items**:

```
[student@workstation todo-rest]$ curl \
http://localhost:8080/todo-rest/api/items
```

Look for the **Item** created in the previous step in the output:

```
[{"id":1,"description":"Complete chapter 6 lab","done":false,"user":
{"id":1,"username":"Guest"}}]
```

- 11.2. Delete the new **Item** using the following DELETE HTTP request with the correct **Item** ID:

```
[student@workstation todo-rest]$ curl -X DELETE \
http://localhost:8080/todo-rest/api/items/1
```

- 11.3. Run the following command to verify that the Item was deleted:

```
[student@workstation todo-rest]$ curl http://localhost:8080/todo-rest/api/items
```

The output displays that the Item is no longer in the database.

**12.** Cleanup and grade.

- 12.1. To verify that you have successfully deployed the application, open a new terminal window on your workstation VM and run the following command to grade this lab:

```
[student@workstation ~]$ lab todo-rest grade
```

If you see failures after running the command, look at the errors, troubleshoot the deployment, and fix the errors. Rerun the grading script and verify that it passes.

- 12.2. In the terminal window where you ran the Maven command to deploy the application, run the following command to undeploy the application from EAP:

```
[student@workstation todo-rest]$ mvn wildfly:undeploy
```

- 12.3. Right-click on the todo-rest project in the **Project Explorer**, and select **Close Project** to close this project.

- 12.4. Right-click **Red Hat JBoss EAP 7.0** in the **Servers** tab, and then click **Stop** to stop the EAP instance.

This concludes the lab.

# Summary

---

In this chapter, you learned:

- JAX-RS is the Java API for creating lightweight RESTful web services.
- Implementing a web service layer allows developers to abstract the front end layer and create an application comprised of many loosely coupled components.
- A JAX-RS RESTful web service consists of one or more classes utilizing the JAX-RS annotations to create a web service.
- The `@ApplicationPath` annotation sets the base URI for the web service.
- The HTTP protocol defines several methods through which the protocol enacts different actions. The client is responsible for specifying the type of request in addition to the path of the request.
- The `@PathParam` annotation maps a variable from a path to the Java method parameter.
- The `request()` method on the `WebTarget` class enables developers to define the type of HTTP request to make to a REST endpoint.
- An HTTP status code of `200` indicates that the request was successful.



## Chapter 7

# Implementing Contexts and Dependency Injection

### Overview

**Goal** Describe typical use cases for using CDI, and successfully implement it in an application.

**Objectives**

- Describe resource injection, dependency injection, and the differences between them.
- Apply scopes to beans appropriately.

**Sections**

- Contrasting Dependency Injection and Resource Injection (and Guided Exercise)
- Applying Contextual Scopes (and Guided Exercise)

**Lab** Implementing Contexts and Dependency Injection

# Contrasting Dependency Injection and Resource Injection

## Objectives

After completing this section, students should be able to:

- Describe dependency injection.
- Describe how to create a qualifier.
- Use a producer to inject non-beans.

## Understanding Context Dependency Injection

The *Contexts and Dependency Injection* (CDI) specification is one of many subordinate specifications within the Java EE specification. While CDI was introduced in Java EE 6, the concepts behind CDI have been around in various frameworks including Spring, Google Guice, and others. The Java Community Process introduced Java Specification Request 299 in its final form in December 2009. JSR 346 formally defines CDI for the Java EE 7 platform. This means that every application server that is certified as Java EE 7 compliant, such as JBoss EAP, must support contexts and dependency injection natively.

There are two primary parts of CDI: *Contexts* and *Dependency Injection*. As it relates to CDI, contexts refer to the ability to define applications by data scope, which is discussed in detail in the next section. Dependency Injection, as specified by CDI, is the process by which instances of objects can be automatically instantiated into other application objects in a type-safe manner. The decision of which particular implementation of an object will be injected can be delayed until the time of application deployment. In other frameworks, injection is based upon string matching. CDI improves upon this through typed injection, where the types are checked at compile time. Including type safety exposes injection errors earlier in the development lifecycle and makes debugging easier.

One of the major benefits of dependency injection (DI) is loose coupling of application components. For instance, client and server components are loosely coupled because several different versions of the server can be injected into the client. The client works with an interface and is oblivious to which server it is talking to. Taking advantage of the deploy time injection, it is possible to use specific objects for different types of environments, such as production and test environments. For example, you can inject a production or test datasource depending upon your deployment environment.

CDI is similar to using resource injection to inject resources such as a `@PersistenceContext` and a `persistence.xml` file. Both approaches create resource dependencies managed by the container, and both loosely couple application components. They do, however, differ in several important ways. Because resource injection uses the JNDI name to inject a resource, resource injection is not type safe like CDI. CDI is type safe because objects are instantiated based on type. Further, CDI is able to inject regular Java classes directly, while resource injection cannot inject regular classes and instead refers to the resource by JNDI names.

## Using Dependency Injection

CDI is not automatically active in a web application, EJB, or Java library (JAR) because it would be inefficient for the container to scan every application and every library. To enable CDI in

web applications, put an empty file named `beans.xml` in the `WEB-INF` directory. For JAR files, including those containing EJBs, put the `beans.xml` file in the `META-INF` directory. The `beans.xml` marker file does not have to contain anything to activate CDI.

There is no special declaration or annotation necessary for a bean to participate in CDI. This is in contrast with an EJB, which requires an annotation marking its type as `@Stateless`, `@MessageDriven`, and so on.

To inject an instance of a bean into the instance variable of another class, use the `@Inject` annotation. When the container scans the annotated class at deployment time, it attempts to find a single bean matching the bean type that is annotated. If the container finds more than one match, it creates an ambiguous dependency error. The `@Inject` annotation is typically used either with a member declaration or in the Java class's constructor parameter. The following is an example utility class named `EmailValidator` with a public `checkEmail` method:

```
public class EmailValidator {

 public boolean checkEmail(String email) {
 ...
 }

}
```

To leverage this class and method with CDI, inject an instance of the class using the following injection code:

```
public class Form {
 ...
 @Inject
 private EmailValidator emailValidator;

 public void submitForm(){
 ...
 emailValidator.checkEmail(email);
 ...
 }
}
```

Notice that a new `EmailValidator` class is never declared. Using injection, the `EmailValidator` class is automatically instantiated by the container.

## Using Qualifiers

A qualifier is a custom annotation that can be applied to a bean class at the site of injection to define which bean type to actually inject. This is useful when a bean injects an interface, but multiple implementations exist using the same interface. When this type of ambiguous injection occurs, the container is not able to choose which implementation to inject. Qualifiers resolve this ambiguity by allowing users to create custom qualifier annotations to indicate which implementation the container should use.

Qualifiers are defined using the following template:

```
@Qualifier
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER}) public @interface qualifier-name { }
```

For example, the following is a qualifier that creates the annotation **@SlowBike**:

```
@Qualifier
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER}) public @interface SlowBike { }
```

After creating that qualifier, annotate the class header with **@SlowBike**:

```
@SlowBike
public class Moped extends Bike implements Vehicle<Motorized> { }
```

Now when a bean injects the **Vehicle** interface and uses the **SlowBike** annotation, the container automatically instantiates an instance of the **Moped** class implementation:

```
public class Inventory {
 @Inject
 @SlowBike
 private Bike myBike;
 ...
}
```

All CDI beans have a qualifier by default. If one is not specified, then the qualifier is **@Default**. Additionally, if a bean is annotated explicitly with the **@Default** annotation, then that bean serves as the default implementation when no other qualifier is specified at the injection point.

## Using Producers

A major advantage of using CDI is that implementation decisions can be delayed past compile time. To facilitate this behavior, *Producers* provide the ability for implementation decisions to be made at runtime using customizable logic to determine how to make those decisions. Producers are methods or object attributes that produce an injectable object. The advantage of producers is that the annotation makes non-bean objects injectable.

The **@Produces** annotation is attachable to the following objects:

- Managed beans
- Primitives, such as **int** and **long**
- Parameterized types such as **Set<String>**
- Raw types such as **Set**

The following is an example of a method annotated with **@Produces** and the injection of the **PaymentStrategy**:

```
@Produces @Preferred public PaymentStrategy getStrategicPaymentStrategy() { ... }
```

```
public class CalculateInterestBean {
 ...
 @Inject @Preferred PaymentStrategy strategy;
 ...
}
```

The previous example shows a producer being used on a method declaration. It is very common to use a qualifier on a producer method to distinguish the type of object available for injection. Combining qualifiers and producers allows developers to provide multiple producer methods and then use ambiguous injections with a qualifier to distinguish which producer method should be used. In the previous example, the object is injected with the qualifier `@Preferred`.

When annotating an attribute in a Java class with `@Produces`, the attribute can then be injected into an attribute in any managed bean. This is useful for declaring and using Java EE resources, such as datasources and loggers. Like producer methods, producer fields are often annotated with a qualifier.

```
@Produces @AccountsDB Connection connection = new Connection(userID, password);
```

## Comparing EJB and CDI

Differentiating EJB and CDI is important because there is an overlap in features between the two specifications. In Java EE 7 applications built to run on JBoss EAP, it is common for developers to use both technologies in conjunction with each other. All EJBs are CDI beans, and therefore have access to dependency injection and are eligible to be injected themselves.

The EJB specification builds upon the CDI specification and provides even more functionality, distinguishing between stateless and stateful beans. EJBs also provide other functionality such as concurrency features, bean pooling, security, as well as others that are not included in CDI. When creating a bean, it is a good practice to not use an EJB if the features of an EJB are not required. Instead, use CDI for managing contexts and dependency injection.



### References

#### JSR 346 Contexts and Dependency Injection for Java

<https://www.jcp.org/en/jsr/detail?id=346>



### References

Further information is available in the CDI chapter of the *Development Guide* for Red Hat JBoss EAP:

<https://access.redhat.com/documentation/en/red-hat-jboss-enterprise-application-platform/7.0/>

## ► Guided Exercise

# Dependency Injection

In this exercise, you will create and inject a utility class with a qualifier.

## Outcomes

You should be able to create a qualifier and inject a bean.

## Before You Begin

Open a terminal window on the **workstation** VM and run the following command to download the lab files required for this guided exercise.

```
[student@workstation ~]$ lab dependency-injection setup
```

## Steps

- 1. Import the dependency-injection project into the JBoss Developer Studio IDE (JBDS).
  - 1.1. Start JBDS by double-clicking the JBDS icon on the **workstation** VM desktop.
  - 1.2. In the Eclipse Launcher window, enter **/home/student/JB183/workspace** in the **Workspace** field, and then click **Launch**.
  - 1.3. In the JBDS menu, click **File > Import** to open the **Import** wizard.
  - 1.4. On the **Select** page, click **Maven > Existing Maven Projects**, and then click **Next**.
  - 1.5. In the **Maven projects** page, click **Browse** to open the **Select root folder** window. Navigate to the **/home/student/JB183/labs/** directory. Select the **dependency-injection** folder, and then click **OK**.
  - 1.6. On the **Maven projects** page, click **Finish**.
  - 1.7. Watch the JBDS status bar to monitor the progress of the import operation. It may take a few minutes to download all the required dependencies.
- 2. Create a new interface called **NameUtil.java** with the method **sanitizeName(String name)**.
  - 2.1. In the **Project Explorer** tab in the left pane of JBDS, right-click **dependency-injection > Java Resources > src/main/java > com.redhat.training.util** and click **New > Interface**.
  - 2.2. Enter **NameUtil** as the name of the interface and click **Finish**.
  - 2.3. Add the following code to the new interface for the **sanitizeName(String name)** method:

```
package com.redhat.training.util;

public interface NameUtil {

 public String sanitizeName(String name);

}
```

- 2.4. Save your changes by pressing **Ctrl+S**.
- 3. Create two new classes in the `com.redhat.com.training.util` package that each implement the `NameUtil` interface.
- 3.1. In the **Project Explorer** tab in the left pane of JBDS, right-click `dependency-injection > Java Resources > src/main/java > com.redhat.training.util` and click **New > Class**.
  - 3.2. Enter `AllCaps` as the name of the class and click **Finish**.
  - 3.3. Update the class header to implement the `NameUtil` interface:

```
package com.redhat.training.util;

public class AllCaps implements NameUtil{

}
```

- 3.4. Hover over the `AllCaps` class name and click **Add unimplemented methods** to create the `sanitizeName(String name)` method and to remove the error. The resulting code looks like the following:

```
@Override
public String sanitizeName(String name) {
 // TODO Auto-generated method stub
 return null;
}
```

3.5. Update the `return` statement to return the name string in all capital letters:

```
return name.toUpperCase();
```

- 3.6. Save your changes by pressing **Ctrl+S**.
- 3.7. In the **Project Explorer** tab in the left pane of JBDS, right-click `dependency-injection > Java Resources > src/main/java > com.redhat.training.util` and click **New > Class**.
- 3.8. Enter `TitleCase` as the name of the class and click **Finish**.
- 3.9. Update the class header to implement the `NameUtil` interface:

```
package com.redhat.training.util;

public class TitleCase implements NameUtil{
}
```

- 3.10. Hover over the `TitleCase` class name and click **Add unimplemented methods** to create the `sanitizeName(String name)` method and to remove the error. The resulting code looks like the following:

```
@Override
public String sanitizeName(String name) {
 // TODO Auto-generated method stub
 return null;
}
```

- 3.11. Update the `return` statement to return the `name` string with the first letter capitalized:

```
return name.substring(0,1).toUpperCase() + name.substring(1);
```

- 3.12. Save your changes by pressing `Ctrl+S`.

- ▶ 4. Add a `@PostConstruct` method to each new utility bean that prints a log statement to declare when the bean is injected.

- 4.1. In the `AllCaps.java` class, add the following `@PostConstruct` method and `javax.annotation.PostConstruct` import:

```
import javax.annotation.PostConstruct;

...
@PostConstruct
public void printTitle(){
 System.out.println("****AllCaps PostConstruct****");
}
```

- 4.2. Save your changes by pressing `Ctrl+S`.

- 4.3. In the `TitleCase.java` class, add the following `@PostConstruct` method and `javax.annotation.PostConstruct` import:

```
import javax.annotation.PostConstruct;

...
@PostConstruct
public void printTitle(){
 System.out.println("****TitleCase PostConstruct****");
}
```

- 4.4. Save your changes by pressing **Ctrl+S**.
- 5. Inject the `NameUtil` interface into the `PersonService.java` class and use the `sanitizeName(String name)` method prior to persisting the name into the database.
- 5.1. In the **Project Explorer** tab in the left pane of JBDS, click **dependency-injection > Java Resources > src/main/java > com.redhat.training.ejb** and expand it. Double-click the `PersonService.java` file.
  - 5.2. After the class header, add the following code to inject the `NameUtil` interface into the `PersonService` class:

```
...
public class PersonService {

 @Inject
 private NameUtil nameUtil;
...
}
```

- 5.3. Add the following line to the `hello()` method to sanitize the name input before persisting the `Person` into the database and outputting the name:
- ```
//TODO sanitize name
name = nameUtil.sanitizeName(name);
```
- 5.4. Save your changes by pressing **Ctrl+S**. Notice that the warning at the injection point says **Multiple beans are eligible for injection to the injection point**. This is because the interface does not have a qualifier to indicate which implementation to use.
- 6. Create a new qualifier and use the qualifier on the utility class to resolve the ambiguous injection point.
- 6.1. In the **Project Explorer** tab in the left pane of JBDS, right-click **dependency-injection > Java Resources > src/main/java > com.redhat.training.util** and click **New > Other....**
 - 6.2. Enter **Qualifier** in the search box. Select **Qualifier Annotation**, and then click **Next**.
 - 6.3. Enter **Title** in the **Name** field and click **Finish**.
 - 6.4. In the `com.redhat.training.util.TitleCase` class, add the qualifier to the class header:

```
@Title
public class TitleCase implements NameUtil{
...
}
```

- 6.5. Save your changes by pressing **Ctrl+S**.
- 6.6. Return to the `PersonService.java` and notice that the warning is no longer present.

- 7. Start EAP by selecting the **Servers** tab in the bottom pane of JBDS. Right-click the server **Red Hat JBoss EAP 7.0 [Stopped]**, and click the green button to start the server. Watch the **Console** until the server starts and displays the message:

```
INFO [org.jboss.as] (Controller Boot Thread) WFLYSRV0025: JBoss EAP 7.0.0.GA  
(WildFly Core 2.1.2.Final-redhat-1) started
```

- 8. Deploy the application on JBoss EAP using Maven by running the following commands:

```
[student@workstation ~]$ cd /home/student/JB183/labs/dependency-injection  
[student@workstation dependency-injection]$ mvn wildfly:deploy
```

- 9. Test the application.

- 9.1. Use a web browser in the **workstation** VM to navigate to `http://localhost:8080/dependency-injection/` to access the dependency-injection application.
- 9.2. In the text box, enter **shadowman** and click **Submit**. The server returns the following:

```
Hello SHADOMAN! ...
```

Because no qualifier is specified on the injection point of the `NameUtil`, the bean that has the `@Default` annotation is used.



Note

If no qualifier is specified, the class assumes the `@Default` qualifier.

- 9.3. In the EAP server logs, notice that the post construct method output for the `AllCaps` class occurs only after clicking submit, but before the method of the object is actually used:

```
***AllCaps PostConstruct***  
###Before NameUtil method used##  
Hibernate:  
  insert  
  into  
    Person  
...
```

- 10. Update the `NameUtil` injection to use the `@Title` qualifier.

- 10.1. Click the `PersonService.java` file to edit the `PersonService` class in the editor pane.
- 10.2. Add the following line to change the implementation for the `NameUtil` and import the `com.redhat.training.util.Title`:

```
@Inject @Title  
private NameUtil nameUtil;
```

10.3. Save your changes by pressing **Ctrl+S**.

- 11. Re-deploy the application with the following command:

```
[student@workstation dependency-injection]$ mvn wildfly:deploy
```

- 12. Test the application again.

12.1. Use a web browser on the **workstation** VM to navigate to `http://localhost:8080/dependency-injection/` to access the dependency-injection application.

12.2. In the text box, enter **redhat** and click **Submit**. The server returns the following:

```
Hello Redhat! ...
```

12.3. In the EAP server logs, notice that the post-construct method output for the **TitleCase** class is printed:

```
***TitleCase PostConstruct***  
###Before NameUtil method used###  
Hibernate:  
  insert  
  into  
    Person  
...
```

- 13. Undeploy the application and stop EAP.

13.1. In the terminal window where you ran the Maven command to deploy the application, run the following command to undeploy the application:

```
[student@workstation dependency-injection]$ mvn wildfly:undeploy
```

13.2. Right-click on the **dependency-injection** project in the **Project Explorer**, and select **Close Project** to close this project.

13.3. Right-click **Red Hat JBoss EAP 7.0** in the **Servers** tab and then click **Stop** to stop the EAP instance.

This concludes the guided exercise.

Applying Contextual Scopes

Objective

After completing this section, students should be able to apply scopes to beans appropriately.

Understanding the Importance of Scope in Java EE Applications

The Red Hat JBoss EAP container manages the life cycle of all CDI beans defined in applications deployed on the server. When managing the life cycles of stateful beans, JBoss needs to understand when to keep different instances of a bean in memory, and when it is safe to destroy a bean instance. In addition, the container needs to understand when to create a new instance of a bean instead of using an existing one. The amount of time the container keeps a specific instance of an bean alive is known as a bean's scope.

Typically, when dependency injection is used to inject a bean, the injected instance of that bean needs to be able to hold its state for the duration of the user's interaction with the application. The container determines how long the instance should maintain its state based on the bean's scope. Scopes can be defined to be long-lasting, where a bean's state is persisted across many different users and requests. Alternatively, beans can be destroyed and recreated with each incoming request.

There are several scopes available in the CDI specification. To define the scope of a bean, use a class-level annotation as shown in the following example:

```
@ApplicationScoped  
public class HelloCounterBean {
```

The following list summarizes each available scope:

Request Scope

- Uses the `@RequestScoped` annotation.
- The container maintains each instance of a request scoped bean only for a single HTTP request.
- Once the request is completed, the bean instance is destroyed.

Session Scope

- Uses the `@SessionScoped` annotation.
- The container maintains each instance of a session scoped bean for each user's session. A session spans multiple requests but typically expires after a configurable amount of inactivity or the browser cache is cleared.
- Once the session expires, the bean instance is destroyed.

Conversation Scope

- Uses the `@ConversationScoped` annotation.
- The container treats conversation scope similar to session scope in that it holds state associated with a user of the system, and spans multiple requests to the server.
- However, unlike the session scope, the conversation scope holds state associated with a particular web browser tab in a web application (browsers typically share cookies across tabs, like the session cookie, so this is not the case for the session scope).
- The beginning and end of a conversation must be manually demarcated by the application. Conversations span all requests in a single browser tab until the application explicitly ends the conversation or a timeout occurs.
- A single user can potentially have multiple concurrent conversations across multiple tabs.

Application Scope

- Uses the `@ApplicationScoped` annotation.
- The container maintains each instance of an application scoped bean for the entire duration of the application's deployment.
- Only when the container is shut down, or the application is redeployed, is the bean instance destroyed.

Singleton

- Uses the `@Singleton` annotation.
- The container maintains only a single instance of singleton beans. Any bean that injects a singleton gets the same instance, because there is only one.

Naming Beans

Another useful feature of the CDI specification is the ability to give beans simple names. These names provide a mechanism for the beans to be referenced using expression language (EL) found in front-end libraries, such as Java Server Faces (JSF) pages. This functionality provides developers with a simple way to inject beans directly into the UI layer of an application.

To assign a name to a CDI managed bean, use the `@Named` annotation. Optionally, specify the name for the bean explicitly, or the name property defaults to the class name with a lower-case first letter.

For example, the following class is a session scoped bean with the `@Named` annotation:

```
@SessionScoped
@Named
public class Hello implements Serializable{

    private static final long serialVersionUID = 1L;

    private String name
```

To reference the `Hello` bean's attributes with JSF, use the `#{{beanName.attributeName}}` syntax. This references the member variable directly and JSF is able to automatically call the

related getter and setter methods to keep the value in the bean automatically updated with the value from the input field on the form.

The following is a snippet of a JSF page using EL with the named `hello` bean, directly referencing the `name` variable and assigning it to an input field on a JSF form:

```
<h:inputText value="#{hello.name}" id="name" required="true" requiredMessage="Name  
is required"/>
```



References

Further information is available in the CDI chapter of the *Development Guide* for Red Hat JBoss EAP:

<https://access.redhat.com/documentation/en/red-hat-jboss-enterprise-application-platform/7.0/>

► Guided Exercise

Applying Scopes

In this exercise, you will apply different scopes to EJBs and test the effects on an application.

Outcome

You should be able to implement different scopes in EJB classes using annotations.

Before You Begin

Open a terminal window on the **workstation** VM and run the following command to download the files required for this exercise:

```
[student@workstation ~]$ lab apply-scopes setup
```

Steps

- 1. Open JBDS and import the Maven project.
 - 1.1. Open JBDS by double-clicking the JBoss Developer Studio icon on the workstation desktop. Set the workspace to /home/student/JB183/workspace and click OK.
 - 1.2. In the JBDS menu, click **File > Import** to open the **Import** wizard.
 - 1.3. On the **Select** page, click **Maven > Existing Maven Projects**, and then click **Next**.
 - 1.4. In the **Maven projects** page, click **Browse** to open the **Select root folder** window. Navigate to the /home/student/JB183/labs/ directory. Select the **apply-scopes** folder, and then click **OK**.
 - 1.5. On the **Maven projects** page, click **Finish**.
 - 1.6. Watch the JBDS status bar to monitor the progress of the import operation. It may take a few minutes to download all the required dependencies.
- 2. Review the JSF page that the application uses.

Open the **index.xhtml** JSF page by expanding the **apply-scopes** item in the **Project Explorer** tab in the left pane of JBDS, then click **apply-scopes > src > main > webapp** and expand it. Double-click the **index.xhtml** file and then click **Source** at the bottom of the editor.

```
<h:form id="form">
<p class="input">
<h:outputLabel value="Enter your name:" for="name" />
<h:inputText value="#{hello.name}" ❶ id="name" required="true"
requiredMessage="Name is required"/>
</p>
<br class="clear"/>
```

```

<br class="clear"/>
<p class="input">
    <h:commandButton action="#{hello.sayHello()}" value="Submit"
styleClass="btn" />
</p>
<br class="clear"/>
<br class="clear"/>
<h:messages styleClass="messages"/>
<p>Greeted <h:outputText value="#{counter.currentCount}" />② person(s) since last
server restart. </p>
</h:form>

```

- ①** An EJB named `hello` is used to provide the variable name as well as the `sayHello()` method.
- ②** An EJB named `counter` is used to provide the variable `currentCount`.

► 3. Update the PersonService EJB class to be stateless.

- 3.1. Open the PersonService class by expanding the `apply-scopes` item in the Project Explorer tab in the left pane of JBDS, then click `apply-scopes > Java Resources > src/main/java > com.redhat.training.ejb` and expand it. Double-click the `PersonService.java` file.

This EJB class maintains no state. It does not have any member variables that need to be specific to each instance of the class. Each instance can be used interchangeably by the application container.

- 3.2. Update the EJB class to include the `@Stateless` annotation to tell the container that the bean does not have any state and can be managed as such.

```

//TODO mark as a stateless EJB
@Stateless
//TODO assign the name "personService" to this EJB

public class PersonService {

```

- 3.3. Update the EJB class to include the `@Named` annotation to tell the container to make the bean available with the name `personService`.

```

//TODO mark as a stateless EJB
@Stateless
//TODO assign the name "personService" to this EJB
@Named("personService")
public class PersonService {

```

- 3.4. Save the changes to the file using `Ctrl+S`.

► 4. Update the HelloCounterBean to be application scoped and use the name `counter`.

- 4.1. Open the `HelloCounterBean` class by expanding the `apply-scopes` item in the Project Explorer tab in the left pane of JBDS, then click `apply-scopes > Java Resources > src/main/java > com.redhat.training.ejb` and expand it. Double-click the `HelloCounterBean.java` file.

This EJB class does maintain state. The `count` variable must be maintained. In this case, set it to use application scope so that the state of this EJB is maintained throughout the life of the application.

Additionally, because the count can be a single value for the application, only a single instance of the `HelloCounterBean` is required.

- 4.2. Update the EJB class to use the `@ApplicationScoped` annotation to tell the container that the state of the bean should be kept alive throughout the entire life of the application.

```
//TODO make application scoped
@ApplicationScoped
//TODO make a singleton

//TODO assign the name "counter" to this EJB

public class HelloCounterBean {
```

- 4.3. Update the EJB class to use the `@Singleton` annotation to tell the container that only a single instance of the bean should be kept alive throughout the entire life of the application.

```
//TODO make application scoped
@ApplicationScoped
//TODO make a singleton
@Singleton
//TODO assign the name "counter" to this EJB

public class HelloCounterBean {
```

- 4.4. Additionally, recall that the JSF page references this bean using the name `counter`.

Update the EJB class to include the `@Named` annotation to tell the container to make the bean available with the name `counter`.

```
//TODO make application scoped
@ApplicationScoped
//TODO make a singleton
@Singleton
//TODO assign the name "counter" to this EJB
@Named("counter")
public class HelloCounterBean {
```

- 4.5. Save the changes to the file using `Ctrl+S`.

- 5. Update the `Hello` backing bean to be session scoped and use the name `hello`.

- 5.1. Open the `Hello` class by expanding the `apply-scopes` item in the **Project Explorer** tab in the left pane of JBDS, then click **apply-scopes** > **Java Resources** > `src/main/java` > `com.redhat.training.ui` and expand it. Double-click the `Hello.java` file.

This class also maintains state. The `name` variable must be maintained. In this case, set it to use session scope so that the state of this class is maintained throughout the life of each user's session in the application.

- 5.2. Update the class to use the `@SessionScoped` annotation to tell the container that the state of the bean should be kept alive throughout the life of each user's session.

```
//TODO set the scope
@SessionScoped
//TODO assign the name "hello" to this EJB

public class Hello implements Serializable{
```

- 5.3. Additionally, recall that the JSF page references this bean using the name `hello`.

Update the class to include the `@Named` annotation to tell the container to make the bean available with the name `hello`.

```
//TODO set the scope
@SessionScoped
//TODO assign the name "hello" to this EJB
@Named("hello")
public class Hello implements Serializable{
```

- 5.4. Save the changes to the file using **Ctrl+S**.

- 6. Start EAP by selecting the **Servers** tab in the bottom pane of JBDS. Right-click the server **Red Hat JBoss EAP 7.0 [Stopped]**, and click the green button to start the server. Watch the **Console** until the server starts and displays the message:

```
INFO [org.jboss.as] (Controller Boot Thread) WFLYSRV0025: JBoss EAP 7.0.0.GA
(WildFly Core 2.1.2.Final-redhat-1) started
```

- 7. Deploy the application on JBoss EAP using Maven by running the following commands:

```
[student@workstation ~]$ cd /home/student/JB183/labs/apply-scopes
[student@workstation apply-scopes]$ mvn wildfly:deploy
```

Confirm successful deployment in the server log shown in the **Console** tab in JBDS. When the app is deployed, the following appears in the log:

```
INFO [org.jboss.as.server] (management-handler-thread - 9) WFLYSRV0010: Deployed
"apply-scopes.war" (runtime-name : "apply-scopes.war")
```

- 8. Test the application scopes in a browser.

- 8.1. Open <http://localhost:8080/apply-scopes> in a browser on the workstation VM.

- 8.2. Enter **Shadowman** in the text box labeled **Enter your name:** and click **Submit**

The page updates with the message *Hello SHADOWMAN!. Time on the server is: Nov 02 2017 03:55:03 PM*

The counter message updates to *Greeted 1 person(s) since the last server restart.*

- 8.3. Navigate to <http://localhost:8080/apply-scopes> again in the browser on the workstation VM.

Notice that the name **Shadowman** is still in the text field. This field persists across page refreshes. This is because the bean backing the form field is session scoped. Additionally, the counter is still at 1 person, because the counter bean is application scoped.

► 9. Clear the cache and refresh the page.

- 9.1. In Firefox, navigate to `about:preferences` in the URL to access the Firefox preferences page.
- 9.2. Click **Privacy** and then click **clear your recent history**. Click **Clear Now** to force the browser to create a new session.
- 9.3. Refresh the page using **Ctrl+R**.

Notice that the name **Shadowman** is **not** in the text field. This field persists across page refreshes, but not across a new session. However, the counter is still at 1 person because the counter bean is application scoped.

► 10. Update the Hello EJB class to be request scoped.

- 10.1. Open the **Hello** class by expanding the **apply-scopes** item in the **Project Explorer** tab in the left pane of JBDS, then click **apply-scopes > Java Resources > src/main/java > com.redhat.training.ui** and expand it. Double-click the **Hello.java** file.
- 10.2. Update the EJB class to use the **@RequestScoped** annotation to tell the container that the state of the bean should be kept alive throughout the life of each user's request.

```
//TODO set the scope
@RequestScoped
//TODO assign the name "hello" to this EJB
@Named("hello")
public class Hello implements Serializable{
```

- 10.3. Save the changes to the file using **Ctrl+S**.

► 11. Re-deploy the application on JBoss EAP using Maven by running the following commands:

```
[student@workstation apply-scopes]$ mvn wildfly:deploy
```

Confirm successful deployment in the server log shown in the **Console** tab in JBDS. When the app is deployed, the following appears in the log:

```
INFO [org.jboss.as.server] (management-handler-thread - 9) WFLYSRV0010: Deployed "apply-scopes.war" (runtime-name : "apply-scopes.war")
```

► 12. Test the application scopes in a browser.

- 12.1. Open `http://localhost:8080/apply-scopes` in a browser on the **workstation** VM.
- 12.2. Enter **Shadowman** in the text box labeled **Enter your name:** and click **Submit**
The page updates with the message *Hello SHADOWMAN! Time on the server is: Nov 02 2017 03:55:03 PM*

The counter message updates to *Greeted 1 person(s) since the last server restart.*

- 12.3. Navigate to `http://localhost:8080/apply-scopes` again in a browser on the **workstation** VM.

Notice that the name **Shadowman** is cleared from the text field. This field no longer persists across the page the refresh. This is because the bean backing the form field is now request scoped. Additionally, the counter is still at 1 person, because the counter bean is application scoped.

► **13.** Undeploy the application and stop JBoss EAP.

- 13.1. Run the following command to undeploy the application:

```
[student@workstation apply-scopes]$ mvn wildfly:undeploy
```

- 13.2. To close the project, right-click the **apply-scopes** project in the **Project Explorer**, and select **Close Project**.

- 13.3. Right-click the Red Hat JBoss EAP 7.0 server in the **JBDS Servers** tab and click **Stop**.

This concludes the guided exercise.

► Lab

Implementing Contexts and Dependency Injection

In this lab, you will use CDI concepts to complete the To Do List application.

Outcome

You should be able to inject resources and define context for beans.

Before You Begin

Open a terminal window on the workstation VM and run the following command to download the files required for this lab.

```
[student@workstation ~]$ lab cdi-lab setup
```

1. Open JBDS and import the `cdi-lab` project located in the `/home/student/JB183/labs/cdi-lab` directory.
2. In the `Resources.java` class, set the persistence context for the `EntityManager` to be the default persistence and make this context injectable in other classes. Update the `produceLog()` method to make the configured `Logger` injectable.
3. Update the `ItemRepository` and `UserRepository` to be stateless and to use the entity manager with the default persistence context.
4. Update the `ItemService.java` and `UserService.java` class to be stateless and to inject the `Logger` and `EntityManager`.
5. Update the `ItemResourceRESTService` to be request scoped and inject the `UserRepository`, `ItemRepository`, `ItemService`, `UserService`, and the `Logger`.
6. Start JBoss EAP from within JBDS.
7. Build and deploy the application to JBoss EAP using Maven.
8. Test the application in a browser.
 - 8.1. Open Firefox on the workstation VM, and navigate to `http://localhost:8080/cdi-lab` to access the application.
 - 8.2. Add at least two new to-do items using the To Do List application interface and verify that the application works as expected.
9. Open a new terminal window and run the following command to grade the lab:

```
[student@workstation ~]$ lab cdi-lab grade
```

The grading script should report `SUCCESS`. If there is a failure, check the errors and fix them until you see a `SUCCESS` message.

10. Clean up.

- 10.1. Undeploy the application from JBoss EAP using Maven with the following commands:

```
[student@workstation ~]$ cd /home/student/JB183/labs/cdi-lab  
[student@workstation cdi-lab]$ mvn wildfly:undeploy
```

- 10.2. Right-click on the **cdi-lab** project in the **Project Explorer**, and select **Close Project** to close this project.
- 10.3. Right-click on the Red Hat JBoss EAP 7.0 server in the JBDS **Servers** tab and click **Stop**.

This concludes the lab.

► Solution

Implementing Contexts and Dependency Injection

In this lab, you will use CDI concepts to complete the To Do List application.

Outcome

You should be able to inject resources and define context for beans.

Before You Begin

Open a terminal window on the workstation VM and run the following command to download the files required for this lab.

```
[student@workstation ~]$ lab cdi-lab setup
```

1. Open JBDS and import the `cdi-lab` project located in the `/home/student/JB183/labs/cdi-lab` directory.
 - 1.1. To open JBDS, double-click the JBoss Developer Studio icon on the workstation desktop. Leave the default workspace (`/home/student/JB183/workspace`) and click **OK**.
 - 1.2. In the JBDS menu, click **File > Import** to open the **Import** wizard.
 - 1.3. On the **Select** page, click **Maven > Existing Maven Projects**, and then click **Next**.
 - 1.4. In the **Maven projects** page, click **Browse** to open the **Select root folder** window. Navigate to the `/home/student/JB183/labs/` directory. Select the `cdi-lab` folder, and then click **OK**.
 - 1.5. On the **Maven projects** page, click **Finish**.
 - 1.6. Watch the JBDS status bar to monitor the progress of the import operation. It may take a few minutes to download all the required dependencies.
2. In the `Resources.java` class, set the persistence context for the `EntityManager` to be the default persistence and make this context injectable in other classes. Update the `produceLog()` method to make the configured `Logger` injectable.
 - 2.1. Open the `Resources` class by expanding the `cdi-lab` item in the Project Explorer tab in the left pane of JBDS, then click `cdi-lab > Java Resources > src/main/java > com.redhat.training.todo.util` and expand it. Double-click the `Resources.java` file.
 - 2.2. Add the `@PersistenceContext` annotation to the `EntityManager` and the corresponding `javax.persistence.PersistenceContext` import to the `Resources` class:

```
import javax.persistence.PersistenceContext;
```

```
@PersistenceContext
```

```
private EntityManager em;
```

- 2.3. Add the `@Produces` annotation to the `EntityManager` and the corresponding `javax.enterprise.inject.Produces` import to the `Resources` class:

```
import javax.enterprise.inject.Produces;
```

```
@PersistenceContext
```

```
@Produces
```

```
private EntityManager em;
```

- 2.4. Add the `@Produces` annotation to the `produceLog()` method:

```
@Produces
```

```
public Logger produceLog(InjectionPoint injectionPoint) {  
    ...
```

- 2.5. Save the changes to the file using `Ctrl+S`.

3. Update the `ItemRepository` and `UserRepository` to be stateless and to use the entity manager with the default persistence context.

- 3.1. Open the `ItemRepository` class by expanding the `cdi-lab` item in the `Project Explorer` tab in the left pane of JBDS, then click `cdi-lab > Java Resources > src/main/java > com.redhat.training.todo.data` and expand it. Double-click the `ItemRepository.java` file.

- 3.2. Add the `@Stateless` annotation to make the `ItemRepository` class stateless and add the corresponding `javax.ejb.Stateless` import:

```
import javax.ejb.Stateless;
```

```
@Stateless
```

```
public class ItemRepository {
```

- 3.3. Add an `@Inject` annotation to the `EntityManager` instance and add the corresponding `javax.inject.Inject` import:

```
import javax.inject.Inject;
```

```
@Inject
```

```
private EntityManager em;
```

- 3.4. Save the changes to the file using `Ctrl+S`.

- 3.5. Open the `UserRepository` class in the `com.redhat.training.todo.data` package by double-clicking the `UserRepository.java` file.

- 3.6. Add the `@Stateless` annotation to make the `UserRepository` class stateless and add the corresponding `javax.ejb.Stateless` import:

```
import javax.ejb.Stateless;
```

```
@Stateless
public class UserRepository {
```

- 3.7. Add an `@Inject` annotation to the `EntityManager` instance and add the corresponding `javax.inject.Inject` import:

```
import javax.inject.Inject;
```

```
@Inject
private EntityManager em;
```

- 3.8. Save the changes to the file using `Ctrl+S`.

- 4.** Update the `ItemService.java` and `UserService.java` class to be stateless and to inject the `Logger` and `EntityManager`.

- 4.1. Open the `ItemService` class by expanding the `cdi-lab` item in the `Project Explorer` tab in the left pane of JBDS, then click `cdi-lab > Java Resources > src/main/java > com.redhat.training.todo.service` and expand it. Double-click the `ItemService.java` file.

- 4.2. Add the `@Stateless` annotation to make the `ItemService` class stateless and add the corresponding `javax.ejb.Stateless` import:

```
import javax.ejb.Stateless;
```

```
@Stateless
public class ItemService {
```

- 4.3. Add an `@Inject` annotation to the `EntityManager` instance:

```
@Inject
private EntityManager em;
```

- 4.4. Add an `@Inject` annotation to the `Logger` instance:

```
@Inject
private Logger log;
```

- 4.5. Save the changes to the file using `Ctrl+S`.

- 4.6. Open the `UserService` class in the `cdi-lab > Java Resources > src/main/java > com.redhat.training.todo.service` by double-clicking the `UserService.java` file.

- 4.7. Add the `@Stateless` annotation to make the `UserService` class stateless and add the corresponding `javax.ejb.Stateless` import:

```
import javax.ejb.Stateless;
```

```
@Stateless
public class UserService {
```

4.8. Add an `@Inject` annotation to the `EntityManager` instance:

```
@Inject
private EntityManager em;
```

4.9. Add an `@Inject` annotation to the `Logger` instance:

```
@Inject
private Logger log;
```

4.10. Save the changes to the file using `Ctrl+S`.

5. Update the `ItemResourceRESTService` to be request scoped and inject the `UserRepository`, `ItemRepository`, `ItemService`, `UserService`, and the `Logger`.

5.1. Open the `ItemResourceRESTService` class by expanding the `cdi-lab` item in the `Project Explorer` tab in the left pane of JBDS, then click `cdi-lab > Java Resources > src/main/java > com.redhat.training.todo.rest` and expand it. Double-click the `ItemResourceRESTService.java` file.

5.2. Add the `@RequestScoped` annotation to make the `ItemResourceRESTService` class stateless and add the corresponding `javax.enterprise.context.RequestScoped` import:

```
import javax.enterprise.context.RequestScoped;
```

```
@RequestScoped
public class ItemResourceRESTService {
```

5.3. Add an `@Inject` annotation to the `Logger` and add the corresponding `javax.inject.Inject` import:

```
import javax.inject.Inject;
```

```
@Inject
private Logger log;
```

5.4. Add an `@Inject` annotation to the `ItemRepository` instance:

```
@Inject
private ItemRepository repository;
```

5.5. Add an `@Inject` annotation to the `UserRepository` instance:

```
@Inject
private UserRepository userRepo;
```

5.6. Add an **@Inject** annotation to the **ItemService** instance:

```
@Inject
private ItemService itemService;
```

5.7. Add an **@Inject** annotation to the **UserService** instance:

```
@Inject
private UserService userService;
```

5.8. Save the changes to the file using **Ctrl+S**.

6. Start JBoss EAP from within JBDS.

Select the **Servers** tab in JBDS. Right-click on the server entry **Red Hat JBoss EAP 7.0 [Stopped]** and click the green Start option to start the server. Watch the **Console** tab of JBDS until the server starts and you see the message:

```
INFO [org.jboss.as] (Controller Boot Thread) WFLYSRV0025: JBoss EAP 7.0.2.GA
(WildFly Core 2.1.8.Final-redhat-1) started
```

7. Build and deploy the application to JBoss EAP using Maven.

Open a new terminal, and change directory to the **/home/student/JB183/labs/cdi-lab** folder.

```
[student@workstation ~]$ cd /home/student/JB183/labs/cdi-lab
```

Build and deploy the EJB to JBoss EAP by running the following command:

```
[student@workstation cdi-lab]$ mvn clean wildfly:deploy
```

8. Test the application in a browser.

- 8.1. Open Firefox on the **workstation** VM, and navigate to **http://localhost:8080/cdi-lab** to access the application.
- 8.2. Add at least two new to-do items using the To Do List application interface and verify that the application works as expected.

9. Open a new terminal window and run the following command to grade the lab:

```
[student@workstation ~]$ lab cdi-lab grade
```

The grading script should report **SUCCESS**. If there is a failure, check the errors and fix them until you see a **SUCCESS** message.

10. Clean up.

- 10.1. Undeploy the application from JBoss EAP using Maven with the following commands:

```
[student@workstation ~]$ cd /home/student/JB183/labs/cdi-lab  
[student@workstation cdi-lab]$ mvn wildfly:undeploy
```

- 10.2. Right-click on the **cdi-lab** project in the **Project Explorer**, and select **Close Project** to close this project.
- 10.3. Right-click on the Red Hat JBoss EAP 7.0 server in the JBDS **Servers** tab and click **Stop**.

This concludes the lab.

Summary

In this chapter, you learned:

- One of the major benefits of dependency injection (DI) is loose coupling of application components. The client and server components are loosely coupled because several different versions of the server can be injected into the client.
- CDI is able to inject regular Java classes directly, while resource injection cannot inject regular classes and instead refers to the resource by JNDI names.
- To enable CDI in web applications, put the `beans.xml` marker file in the `WEB-INF` directory. For JAR files including those containing EJBs, put the `beans.xml` file in the `META-INF` directory.
- If no qualifier is specified for a bean, then the qualifier defaults to `@Default` and the bean becomes the default implementation for the bean type or types.
- The advantage of producers is that the annotation makes non-bean objects injectable.
- The following scopes are available in CDI:
 - `@RequestScoped`
 - `@SessionScoped`
 - `@ConversationScoped`
 - `@ApplicationScoped`
 - `@Singleton`
- The `@Named` annotation provides a mechanism for EJBs to be referenced using expression language (EL) found in front-end libraries, such as Java Server Faces (JSF) pages.

Chapter 8

Creating Messaging Applications with JMS

Overview

Goal Create messaging clients that send and receive messages using the JMS API.

- Objectives**
- Describe the JMS API and name the objects used in sending and receiving messages.
 - Describe the components that make up the JMS API.
 - Create a JMS client that produces and consumes messages using the JMS API.
 - Create, package, and deploy a message driven bean.

- Sections**
- Describing Messaging Concepts (and Quiz)
 - Describing JMS Architecture (and Quiz)
 - Creating a JMS Client (and Guided Exercise)
 - Creating Message Driven Beans (and Guided Exercise)

Lab Creating Messaging Applications with JMS

Describing Messaging Concepts

Objectives

After completing this section, students should be able to:

- Describe the JMS API
- Identify the objects used to send and receive messages

Understanding Asynchronous Messaging Concepts

Messaging technology or *message-oriented-middleware (MOM)* is an essential tool used by many developers in enterprises around the world. By enabling *asynchronous* processing between *loosely-coupled* systems, messaging allows developers to build applications that are more efficient, more easily scaled, and more reliable.

Messaging solutions use a concept called *queues*, or *destinations*, to facilitate the transfer of data from one application to another. Developers use queues as an intermediary between applications. Queues store a message's data as it is being transferred from *sender* to *receiver*. Using a queue, the sending application (sender) does not need to wait for a response from the receiving application (receiver). Instead, using messaging, the sender places a message on a queue, and the receiver retrieves the message from the queue, and then processes the message.

A practical example of asynchronous communication between applications is a front-end web application for an e-commerce business sending order data to a back-end order fulfillment system. When a customer places an order, the web application does not need the fulfillment system to fill the order. Instead, the application only needs to send a message with the order information to the order queue, which the back-end system can process later. Then the web application returns the customer to an order confirmation screen while the back-end application processes the order.

Another significant benefit of using messaging to facilitate application integration is that messaging allows for loose coupling. Applications using messaging are said to be loosely coupled together because the only requirement for them to communicate is a connection to the destination, and an agreed upon message format. In a messaging system, application components can be written in many different languages as long as the components adhere to the same message format. The flexibility that this clean separation of applications provides also helps support scalability. For instance, in the previous e-commerce application example, when the order queue is piling up, new instances of the order fulfillment back-end application can easily be created and consume messages from the queue simultaneously, enabling much faster data processing.

Using a Queue for Point-to-point Messaging

Point-to-point messaging is when two applications are connected using a *queue*, and each message that a producer sends is received by a single consumer. While there can be multiple producers or multiple consumers connected to a given queue, a single producer and consumer pair processes each message. Application developers frequently use the point-to-point messaging model for the simplicity and scalability it provides.

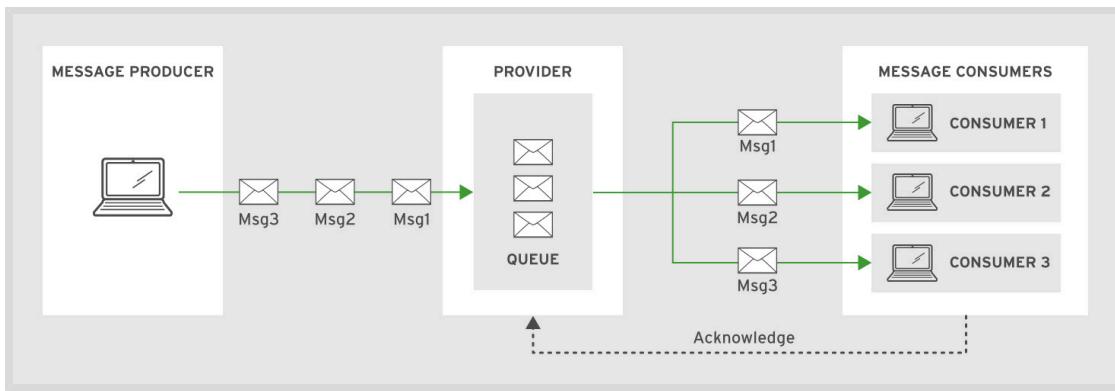


Figure 8.1: A queue in use by a producer and consumer applications

Consumers of a queue use a pull-based model to retrieve new messages. This means that any receiver client typically needs to poll the queue for new messages. Additionally, in the point-to-point model, a queue consumer typically must acknowledge successful processing of the message, or it is put back onto the queue to be retried. Some of this retry behavior depends on the application code and the messaging technology.

Using a Topic for Publish-subscribe Messaging

Publish-subscribe messaging is when multiple applications *subscribe* to a given *topic*, and any message sent to that topic is duplicated and copied to each of the subscribers. Topics are very similar to queues, with the only difference being the number of consumers.

Developers refer to consumers of a topic as *subscribers*. When a new application subscribes to a topic, it then receives any new messages that are sent to that topic, but it does not receive any previous messages sent to the topic prior to subscription.

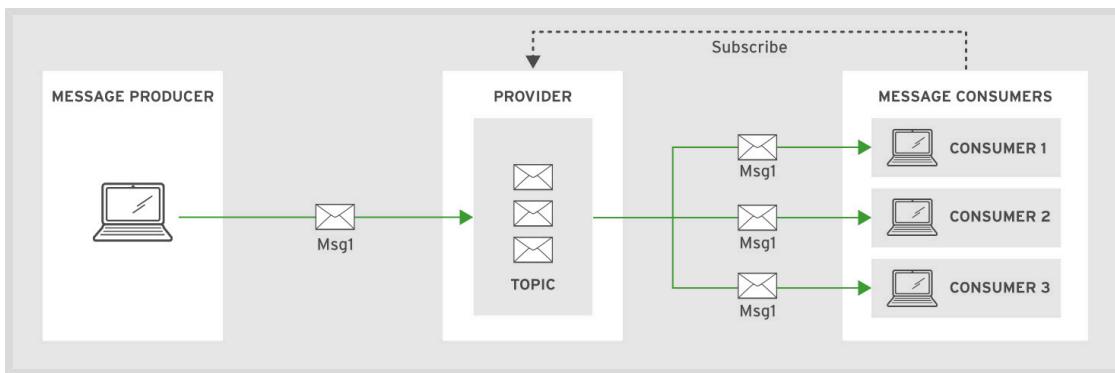


Figure 8.2: A topic in use by a producer and a set of consumer applications

Unlike point-to-point messaging, which uses a pull-based model, publish-subscribe messaging uses a push model. A push model is where the messaging provider is responsible for sending incoming messages to all subscribers.

When an application creates a new subscription to a topic to receive messages, there are two types of subscriptions: *durable* and *non-durable*. During the subscription process, the application must specify if its subscription is durable or non-durable. When using durable subscriptions, if the application disconnects from the topic temporarily, any messages sent to the topic while the app is disconnected are saved, and delivered the next time the durable subscriber reconnects. Alternatively, a non-durable subscription does not save any messages received while the subscriber is disconnected.

Reviewing the JMS Specification in JSR-343

The Java EE specification defines the *Java Message Service (JMS)* API in order to create a standardized method for Java applications to connect to and utilize enterprise messaging solutions. JMS was first introduced in 2002 as JSR-914 and the current version used by Java EE 7, v2.0 is maintained under JSR-343.

Developers use a set of common terminology when developing JMS applications. These terms and their definitions are in the following table:

JMS Concepts

Term	Definition
JMS Client	Java programs or applications that send or receive messages.
Message	Data, typically in a standardized format, sent between applications or clients.
JMS Provider	The messaging component or technology that implements the JMS API. When working with JBoss EAP, a JMS provider is built into the server, or an external provider can be configured.
Administered Object	A preconfigured JMS object defined in the container configuration and is used by an application. Typically, connection factories and destinations are administered objects that are defined at the server level and are made available by the container for use by the deployed applications.
Connection Factory	A Java object used by a client to create new connections to the JMS provider.
Destination	A Java object used by a client to specify the queue or topic where the client sends or receives messages.



References

Further information is available in the JMS chapter of the *Development Guide* for Red Hat JBoss EAP 7 at
<https://access.redhat.com/documentation/en/red-hat-jboss-enterprise-application-platform/>



References

JMS 2.0 JSR

<https://jcp.org/en/jsr/detail?id=343>

► Quiz

Describing Messaging Concepts

Match the items below to their counterparts in the table.

A Java object used by a JMS client to specify where the client sends or receives messages.

A Java object used to provide connections to the JMS provider to the application.

A Java program that sends or receives messages.

A subscription that saves topic messages when the application is not connected.

Destination type used with point-to-point messaging.

Destination type used with publish-subscribe messaging.

The Message-oriented-middleware component used by the JMS application.

Term	Definition
Queue	
Topic	
JMS Provider	
Destination	
JMS Client	
Durable-subscription	
Connection Factory	

► Solution

Describing Messaging Concepts

Match the items below to their counterparts in the table.

Term	Definition
Queue	Destination type used with point-to-point messaging.
Topic	Destination type used with publish-subscribe messaging.
JMS Provider	The Message-oriented-middleware component used by the JMS application.
Destination	A Java object used by a JMS client to specify where the client sends or receives messages.
JMS Client	A Java program that sends or receives messages.
Durable-subscription	A subscription that saves topic messages when the application is not connected.
Connection Factory	A Java object used to provide connections to the JMS provider to the application.

Describing JMS Architecture

Objective

After completing this section, students should be able to describe the components that comprise the JMS API.

Using JMS on JBoss EAP 7 with the Embedded Apache Artemis Message Broker

To provide JMS messaging functionality to its users, JBoss Enterprise Application Platform 7 leverages an Apache ActiveMQ Artemis messaging broker as an embedded JMS provider. This embedded JMS provider allows developers to quickly build, deploy, and integrate messaging applications on JBoss EAP without the cost or overhead of an external messaging provider.

While Apache ActiveMQ Artemis is released and sold as the standalone product Red Hat JBoss AMQ 7, a single embedded broker instance that is based on a similar version of Artemis is included with each instance of JBoss EAP 7. The Artemis implementation replaces the HornetQ broker used by JBoss EAP 6, and is fully backwards compatible.

JBoss EAP 7 deploys and manages Artemis as the `messaging-activemq` subsystem. After installing JBoss EAP, the only profile that enables the messaging subsystem is the `standalone-full.xml` configuration file. It is therefore necessary to either use this configuration file or manually enable the `messaging-activemq` subsystem in a custom `standalone.xml` configuration file.



Note

Using the embedded ActiveMQ Artemis messaging broker requires much of the same configuration and administration as a standalone Artemis broker. While these tasks are not within the scope of this course, more information on these topics and other Artemis-related materials can be found in the Red Hat Training course

JB440: Red Hat JBoss AMQ Administration.

Using Administered Objects for JMS with JBoss EAP 7

A common practice when working with messaging on an application server such as JBoss EAP 7 is to maintain JMS destinations and connection factories administratively in the server configuration rather than in the application code itself. The reason for this is that the connection factory and destination objects typically include environment-specific configuration, such as hosts and ports, as well as potentially sensitive information like user names or passwords.

Additionally, depending on the JMS provider being used, the necessary configuration parameters to connect to that provider can change dramatically. Using administered objects simplifies the application code by providing a layer of abstraction to extract these provider-specific configuration away from the developer.

JMS clients access these administrative objects using the JMS API interfaces that are JMS provider-agnostic, allowing a JMS client to run with potentially many different JMS providers with little to no code modifications when switching.

When using JBoss EAP 7, JMS-administered objects are created as part of the `messaging-activemq` subsystem. The following snippet from the `standalone-full.xml` includes some example administered objects:

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:1.0">
    <server name="default">
        ...Some configuration omitted...
        <jms-queue name="ExpiryQueue" entries="java:/jms/queue/ExpiryQueue"/>
        <jms-queue name="DLQ" entries="java:/jms/queue/DLQ"/>
        <jms-queue name="helloWorldQueue" entries="queue/helloWorldQueue java:jboss/
jms/queue/helloWorldQueue"/>
        <jms-queue name="TodoListQueue" entries="queue/TodoListQueue java:jboss/jms/
queue/TodoListQueue"/>①
        <connection-factory name="InVmConnectionFactory" entries="java:/
ConnectionFactory" connectors="in-vm"/>②
        <connection-factory name="RemoteConnectionFactory" entries="java:jboss/
exported/jms/RemoteConnectionFactory" connectors="http-connector"/>
        <pooled-connection-factory name="activemq-ra" transaction="xa"
entries="java:/JmsXA java:jboss/DefaultJMSConnectionFactory" connectors="in-vm"/>
    </server>
</subsystem>
```

- ①** This defines a JMS queue as an administered object with the JNDI names `queue/TodoListQueue`, and `java:jboss/jms/queue/TodoListQueue`.
- ②** This defines a JMS connection factory as an administered object with the JNDI name `java:/ConnectionFactory`.



Note

Creating administered objects for JBoss EAP 7 is not within the scope of this course. More information on this topic and other EAP-related topics can be found in the Red Hat Training course **JB248: JBoss Administration I**

Reviewing the Components of a JMS Client

There are a number of components of the JMS API that make up a JMS client. Some of these components are provided as administrated objects, and some must be created programmatically by the developer using the JMS API. The following list summarizes the high-level components that are necessary to build a JMS client:

ConnectionFactory

A JMS connection factory is an administered object that a client uses to create a connection to the JMS provider. Typically, the connection factory includes the necessary connection or authentication parameters predefined by a server administrator. Each connection factory is an instance of the `ConnectionFactory`, `QueueConnectionFactory`, or `TopicConnectionFactory` Java interfaces.

Typically, the managed connection factory provided by the application container is injected into a JMS client using the `@Resource` CDI annotation.

```
@Resource(lookup = "java:jboss/ConnectionFactory")
private static ConnectionFactory connectionFactory;
```

It is also possible to do the JNDI lookup manually if an administered object is not defined:

```
final Properties env = new Properties();
env.put(Context.INITIAL_CONTEXT_FACTORY, INITIAL_CONTEXT_FACTORY);
env.put(Context.PROVIDER_URL, System.getProperty(Context.PROVIDER_URL,
    PROVIDER_URL));
env.put(Context.SECURITY_PRINCIPAL, userName);
env.put(Context.SECURITY_CREDENTIALS, password);
namingContext = new InitialContext(env);

// Perform the JNDI lookups
String cfString = System.getProperty("connection.factory",
    DEFAULT_CONNECTION_FACTORY);
ConnectionFactory connectionFactory = (ConnectionFactory)
    namingContext.lookup(cfString);
```

Destination

A JMS destination is an administered object that a client uses to specify the target address for messages it sends, or the source address where it should receive messages. Depending on the type of messaging, a destination can either be a queue or a topic. Some JMS clients even interact with multiple destinations. Similar to the connection factory, the destination is typically injected into the client using the `@Resource` CDI annotation or using a JNDI lookup to find the administered object.

```
@Resource(lookup = "jms/MyQueue")
private static Queue queue;

@Resource(lookup = "jms/MyTopic")
private static Topic topic;
```

Connection

A JMS connection is the virtual connection to the JMS provider, and is typically created in the client by using the JMS API to create a JMS context. The implementation of the connection does not affect the JMS client code, and is entirely handled in the configuration of the connection factory by the server administrator.

Session

A JMS session is a single-threaded resource for either sending or receiving messages, and it is created by a connection. The `Session` class is typically what creates the message listener, message producer, and message objects. JMS sessions also provide the transactional behavior, allowing JMS clients to provide local transactions and group a set of message actions, either sending or receiving, into a single unit of work that can be committed on success or rolled-back on failure.

Context

A JMS context is the combination of a session and a connection, providing both a direct connection to the JMS provider as well as a single-threaded class that can be used to send and receive messages to the provider. When writing a JMS client, typically a developer creates a `JMSContext` object from the `ConnectionFactory` that was injected.

```
JMSContext context = connectionFactory.createContext();
```

Message Producer

A JMS message producer object is created by a JMS context, and can be used to send messages to a destination. The `JMSProducer` interface is provided by the JMS v2.0 API for representing message producers. The JMS context provides a method to create a producer using a context:

```
JMSProducer producer = context.createProducer();
```

Developers can use a simple method call to send a message to a destination:

```
producer.send(dest, message);
```

Message Consumer

A JMS message consumer object is created by a JMS context, and can be used to receive messages from a destination synchronously. The `JMSConsumer` interface is provided by the JMS v2.0 API for representing message consumers. The JMS context provides a method to create a consumer that is connected to a destination using a context:

```
JMSConsumer consumer = context.createConsumer(dest);
```

Developers can use a simple method call to receive messages:

```
Message m = consumer.receive();
```

The `receive()` method takes an optional timeout parameter defining how long it should block execution before returning `null`. To enable asynchronous delivery, a `MessageListener` or a Message Driven Bean must be used instead. This is covered in detail in a later section.

Understanding the Components of a JMS Message

JMS messages use a basic format standardized to allow messages to be flexible enough to suit most use cases, as well as provide some potential for compatibility with non-JMS messaging technologies. The basic structure of a JMS message is three parts: *headers*, *properties*, and a *body*.

Headers

JMS headers are used by both clients and providers to route and classify messages. Headers are key-value pairs, but there is a predetermined set of possible keys defined by the JMS specification. The `Message` interface provides methods to get and set all of the possible header values.

The following table summarizes the JMS header fields and how application developers and JMS providers typically use them:

JMS Header Fields

Header Name	Description
<code>JMSMessageID</code>	The unique identifier for each message sent by a provider, set automatically.
<code>JMSDestination</code>	The destination where the message is being sent.

Header Name	Description
JMSPriority	The priority level of the message. This can be used to reorder messages and advance more important messages to the top of the queue.
JMSDeliveryMode	The persistence settings of the message, controlling how the message is handled by the provider.
JMSDeliveryTime	The timestamp when the message was delivered by the JMS provider.
JMSTimeStamp	The timestamp when the message is given to the JMS provider.
JMSExpiration	The time when the message should be considered expired.
JMSCorrelationID	The ID of another message, used to relate two messages, typically set by the client.
JMSReplyTo	The destination where replies to messages can be sent.
JMSRedelivered	The status of whether or not the message is currently a re-delivery.

Most of the JMS headers are set automatically by the send method of the JMS client, but a few of the headers are intended to be set by the client manually and are left blank if not set. These client-specific headers include `JMSReplyTo`, `JMSCorrelationID`, and `JMSType`.

Properties

Properties provide the ability to set additional values on a JMS message that are not provided by standard headers. Similar to headers, properties are key-value pairs, with no limitations on the possible keys a developer can use. Properties must be set prior to sending a message and are read-only on JMS messages received by a client.

The `Message` interface provides the following methods to get and set properties specific to all of the primitive types, like `int` and `double`. The `Object` superclass is also supported which allows properties to be of any type. There is also a `getPropertyNames()` method that returns an `Enumeration` of all the potential property names for a given JMS message.

Body

The JMS API includes six different types of message based on the object type of the message body. This allows developers to send and receive different types of data easily by using a variety of Java objects to represent the body of the message, which allows for greater flexibility. The following table summarizes the six types and their body content:

JMS Message Body Types

Type	Body
Message	An empty body. This type of message is only headers and properties and is used when a message body is not needed by the application.
BytesMessage	A stream of bytes, typically used to encode a body to match an existing message format.
StreamMessage	A stream of Java primitives, to be read sequentially.

Type	Body
TextMessage	A Java <code>String</code> object, which can include JSON, or XML data.
ObjectMessage	Any serializeable Java object.
MapMessage	A set of key-value pairs, where the keys are <code>String</code> objects and the values are Java primitives.

The JMS API provides methods on the `Context` to create each type of message. The following shows an example of creating a `TextMessage`:

```
TextMessage hello = context.createTextMessage();
hello.setText("Hello World!");
context.createProducer().send(hello);
```



References

Further information is available in the JMS chapter of the *Development Guide* for Red Hat JBoss EAP 7; at
<https://access.redhat.com/documentation/en/red-hat-jboss-enterprise-application-platform/>

► Quiz

Describing JMS Architecture

Match the items below to their counterparts in the table.

Connection

Connection Factory

Consumer

Context

Destination

Producer

Session

Description	JMS Component
An administered object used by clients to set the target or source address for messages.	
An administered object that a client uses to create a connection to the JMS provider.	
A single-threaded resource for either sending or receiving messages created by a connection.	
The combination of a session and a connection, providing both a connection to the JMS provider as well as a single-threaded class used to send and receive messages to the provider.	
The virtual connection to the JMS provider, typically created in the client by first using the JMS API to create a JMS context.	
Created by a JMS context and used to send messages to a destination.	
Created by a JMS context and used to receive messages from a destination synchronously.	

► Solution

Describing JMS Architecture

Match the items below to their counterparts in the table.

Description	JMS Component
An administered object used by clients to set the target or source address for messages.	Destination
An administered object that a client uses to create a connection to the JMS provider.	Connection Factory
A single-threaded resource for either sending or receiving messages created by a connection.	Session
The combination of a session and a connection, providing both a connection to the JMS provider as well as a single-threaded class used to send and receive messages to the provider.	Context
The virtual connection to the JMS provider, typically created in the client by first using the JMS API to create a JMS context.	Connection
Created by a JMS context and used to send messages to a destination.	Producer
Created by a JMS context and used to receive messages from a destination synchronously.	Consumer

Creating a JMS Client

Objective

After completing this section, students should be able to create a JMS client that produces and consumes messages using the JMS API.

Accessing Administered Objects in a JMS Client

In any enterprise that uses messaging-oriented-middleware, system administrators and sometimes developers, are responsible for configuring application servers to have the necessary administered objects to support messaging. These administered objects contain all the relevant connection configuration required to communicate with the JMS provider. When using Red Hat JBoss EAP 7, the JNDI names for these objects is found in the `standalone.xml` configuration file. Removing these connection details from the application code makes the application less brittle and easier to move from one environment to another.

After defining the administered objects, these objects are available directly from the JMS client code through techniques such as dependency injection. For example, applications use JNDI to find the destination and connection factory objects that are needed to create a `JMSContext`.

The `@Resource` annotation directly injects a destination object, such as a `Queue` or `Topic`, as well as a `ConnectionFactory` object using only the JNDI name that is defined in the server configuration. The following example shows the `@Resource` annotation to inject a `Queue` object:

```
@Resource(mappedName = "java:jboss/jms/queue/helloworldQueue")
private Queue helloworldQueue;
```

The `@Resource` annotation is also able to inject a `ConnectionFactory`, as shown in the following example:

```
@Resource(mappedName = "java:comp/DefaultJMSConnectionFactory")
private static ConnectionFactory connectionFactory;

private JMSContext context;

@PostConstruct
public void init(){
    context = connectionFactory.createContext();
}
```

After using the `@Resource` annotation to inject it, the `ConnectionFactory` is used to create the `JMSContext` object. The `JMSContext` creates other JMS objects, such as messages, producers, or consumers.

Alternatively, a `JMSContext` can be created using an `@Inject` annotation. This approach eliminates some unnecessary code required to create the `JMSContext` when the connection factory requires no further customization in the application code. There are two options: specify

the JNDI name of the connection factory to use when creating the `JMSContext`, or use the default connection factory for the application server.

To use the default connection factory for the application server, use the `@Inject` annotation as shown in the following example before declaring the `JMSContext`:

```
@Inject  
private JMSContext context;
```

If a specific connection factory other than the default is required, use the `@JMSSConnectionFactory` annotation in conjunction with the `@Inject` annotation as shown in the following example:

```
@Inject  
@JMSSConnectionFactory("jms/MyConnectionFactory")  
private JMSContext context;
```

Using the Message Producer Interface to Send Messages

A message producer is a type of JMS object created by a `JMSContext` object that sends JMS messages to a destination. The `JMSProducer` interface, introduced in JMS v2.0, encapsulates the functionality of message producers. The producer class is lightweight and doesn't consume a lot of system resources when being created or when in use. For this reason, a single instance of `JMSProducer` can be reused to send multiple messages, or the producer can be created for each message.

The following example shows creating a `JMSProducer` instance, storing it in a variable, and then using that variable to send a message to a destination:

```
JMSProducer producer = context.createProducer(); ①  
  
TextMessage message = context.createTextMessage(msg); ②  
  
producer.send(helloworldQueue, message); ③
```

- ① Create the `JMSProducer` instance using the `JMSContext` and store it in the `producer` variable.
- ② Use the `JMSContext` to create a new `TextMessage` instance from a `String` object named `msg`.
- ③ Send the message to the `helloworldQueue` destination object using the `producer` variable.

Using the Message Consumer Interface to Receive Messages

A message consumer is a type of JMS object created by a `JMSContext` object that receives JMS messages from a destination synchronously using a method call. The `JMSConsumer` interface, introduced in JMS v2.0, is used to encapsulate functionality of message consumers.

The main functionality of a message consumer is provided by the `receive()` method, which includes an optional `timeout` parameter to set how long an application should wait to receive a message from the destination before timing out. To create a resilient message consumer, be sure to handle any JMS exceptions that occur during message delivery. Use a try-catch block when receiving messages from a destination and handle any relevant JMS exceptions.

The JMS provider may allocate some resources on behalf of a `JMSConsumer`. JMS clients should close consumers when they are not needed, instead of relying on garbage collection, which may not be done in a timely manner, increasing the load the application puts on the system. Typically, the consumer calls the `close()` method in the `finally` block after the consumer is no longer needed.

The following example shows creating a `JMSConsumer` instance, storing the consumer in a variable, and then using that consumer variable to receive a message from a destination:

```
JMSConsumer consumer = context.createConsumer(helloWorldQueue); ①

try {
    TextMessage msg = (TextMessage) consumer.receiveNoWait(); ②
    if(msg != null) {
        System.out.println("Received Message: "+ msg);
        return msg.getBody(String.class);
    }else {
        return null;
    }
}
catch (Exception e) {
    e.printStackTrace();
    return null;
} finally {
    consumer.close(); ③
}
```

- ① Create the `JMSConsumer` instance using the `JMSContext` and pass in the `helloWorldQueue` as the destination, then store it in the `consumer` variable.
- ② Use the `receiveNoWait` to check for a new message on the queue but do not block if no message is available.
- ③ Close the consumer object to release the connection and conserve resources.

Demonstration: Creating a JMS Client

1. Run the following command to prepare files used by this demonstration.

```
[student@workstation ~]$ demo hello-jms setup
```

2. Start JBDS and import the `hello-jms` project.

This project is a simple web app using a JSF page backed by a stateful request-scoped EJB to print a simple welcome message to the user after they submit their name. It also sends a JMS message to a queue each time it greets the user. There is also an option to retrieve the oldest message from the queue and display it on the page.

3. Inspect the `pom.xml` file, and observe the dependency on the JBoss library that the server uses for the JMS specification.

```
<dependency>
  <groupId>org.jboss.spec.javax.jms</groupId>
  <artifactId>jboss-jms-api_2.0_spec</artifactId>
  <scope>provided</scope>
</dependency>
```

4. Update the `JMSClient` EJB class in the `messaging` package to inject a `JMSContext` connected to the embedded Artemis broker and map the `Queue` to the administered object using JNDI:

```
@Startup
@Singleton
public class JMSClient {

    //TODO Map the destination queue to the admin object using JNDI and @Resource
    @Resource(mappedName = "java:jboss/jms/queue/helloworldQueue")
    private Queue helloworldQueue;

    //TODO Inject a JMSContext to get a Connection and Session to the embedded
    broker
    @Inject
    JMSContext context;
    ...
}
```

5. In the `JMSClient` EJB class, finish the `sendMessage` method and create a `MessageProducer` to send each incoming greeting message to the queue:

```
public void sendMessage(String msg) {
    try {
        //TODO Create a JMSProducer
        JMSProducer producer = context.createProducer();

        //TODO Create a TextMessage
        TextMessage message = context.createTextMessage(msg);

        //TODO Send the message
        producer.send(helloworldQueue, message);

        System.out.println("Sent Message: " + msg);
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}
```

6. In the `JMSClient` EJB class, finish the `getMessage` method, create a `MessageConsumer` class and then receive the oldest message from the queue:

```

public String getMessage() {

    //TODO Create a JMS Consumer
    JMSConsumer consumer = context.createConsumer(helloWorldQueue);

    try {
        //TODO receive a message without waiting
        TextMessage msg = (TextMessage) consumer.receiveNoWait();
        if(msg != null) {
            System.out.println("Received Message: "+ msg);
            return msg.getBody(String.class);
        }else {
            return null;
        }
    }
    catch (Exception e) {
        e.printStackTrace();
        return null;
    } finally {
        //TODO close the consumer
        consumer.close();
    }
}

```

7. Start the local JBoss EAP server inside JBDS.
8. Deploy the application to the local JBoss EAP server, and test it in a browser. Run the following command in a terminal window:

```
[student@workstation ~]$ cd /home/student/JB183/labs/hello-jms
[student@workstation hello-jms]$ mvn wildfly:deploy
```

Open <http://localhost:8080/hello-jms/> in your browser, then test the application and make sure it is properly displaying the greeting for each name that is input and that it can retrieve those greetings from the queue correctly.

9. Undeploy the application and stop the server.

```
[student@workstation hello-jms]$ mvn wildfly:undeploy
```



References

Further information is available in the JMS chapter of the *Configuring Messaging* guide for Red Hat JBoss EAP 7; at <https://access.redhat.com/documentation/en/red-hat-jboss-enterprise-application-platform/>

► Guided Exercise

Creating a JMS Client

In this exercise, you will write a JMS client that uses the JMS API and queue located on the embedded Artemis broker in JBoss EAP to send and receive JMS messages.

Outcome

You should be able to use the JMS API and the administered objects provided by JBoss EAP to build an instance of `MessageProducer` and a `MessageConsumer` interfaces to send and receive messages from a queue.

Before You Begin

Open a terminal window on the `workstation` VM and run the following command to download the files required for this lab.

```
[student@workstation ~]$ lab jms-client setup
```

Steps

- 1. Open JBDS and import the Maven project.
 - 1.1. Open JBDS by double-clicking the JBoss Developer Studio icon on the workstation desktop. Set the workspace to `/home/student/JB183/workspace` and click **OK**.
 - 1.2. In the JBDS menu, click **File > Import** to open the **Import** wizard.
 - 1.3. On the **Select** page, click **Maven > Existing Maven Projects**, and then click **Next**.
 - 1.4. On the **Maven projects** page, click **Browse** to open the **Select root folder** window. Navigate to the `/home/student/JB183/labs/` directory. Select the `jms-client` folder, and then click **OK**.
 - 1.5. On the **Maven projects** page, click **Finish**.
 - 1.6. Watch the JBDS status bar to monitor the progress of the import operation. It may take a few minutes to download all the required dependencies.
- 2. Review the JNDI names of the JMS administered objects defined in the configuration of the local JBoss EAP 7 server.
 - 2.1. Open a terminal window on the `workstation` VM and using your preferred text editor, open the EAP configuration file at `/opt/eap/standalone/configuration/standalone-full.xml`.

```
[student@workstation ~]$ less /opt/eap/standalone/configuration/\
standalone-full.xml
```

- 2.2. Scroll down in the file until you see the `urn:jboss:domain:messaging-activemq:1.0` subsystem:

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:1.0">
  <server name="default">
    ...
    <jms-queue name="ExpiryQueue" entries="java:/jms/queue/ExpiryQueue"/>
    <jms-queue name="DLQ" entries="java:/jms/queue/DLQ"/>
    <jms-queue name="helloworldQueue" entries="queue/helloworldQueue java:jboss/jms/
queue/helloworldQueue"/>
    <jms-queue name="TodoListQueue" entries="queue/TodoListQueue java:jboss/jms/
queue/TodoListQueue"/>
    <connection-factory name="InVmConnectionFactory" entries="java:/
ConnectionFactory" connectors="in-vm"/>
    <connection-factory name="RemoteConnectionFactory" entries="java:jboss/exported/
jms/RemoteConnectionFactory" connectors="http-connector"/>
    <pooled-connection-factory name="activemq-ra" transaction="xa" entries="java:/
JmsXA java:jboss/DefaultJMSConnectionFactory" connectors="in-vm"/>
  </server>
</subsystem>
```

Note that the `helloworldQueue` has a JNDI entry of `java:jboss/jms/queue/helloworldQueue`.

- 2.3. Close the text editor and **do not** save any changes to the `standalone-full.xml` file.



Warning

Problems can occur in the exercise if unexpected changes are introduced in the configuration file.

- 3. Configure the JMS context and destination.

- 3.1. Open the `JMSClient` class by expanding the `jms-client` item in the Project Explorer tab in the left pane of JBDS, then click on `jms-client` > Java Resources > `src/main/java` > `com.redhat.training.messaging` and expand it. Double-click the `JMSClient.java` file.
- 3.2. Use the `@Inject` annotation to inject the default `JMSContext`, which provides a connection to the embedded Artemis broker running on the local JBoss server.

```
//TODO Inject a JMSContext to get a Connection and Session to the embedded broker
@Inject
JMSContext context;
```

- 3.3. Use the `@Resource` annotation to inject the `helloworldQueue` destination:

```
/TODO Map the destination queue to the admin object using JNDI and @Resource
@Resource(mappedName = "java:jboss/jms/queue/helloworldQueue")
private Queue helloworldQueue;
```

Ensure that the `mappedName` attribute is correctly set to the JNDI name of the queue.

- 3.4. Save the changes to the file using **Ctrl+S**.

▶ **4.** Create a JMS producer that puts messages onto the `helloWorldQueue`.

- 4.1. Use the `createProducer` method that is provided by the `JMSPContext` interface to build an instance of `MessageProducer` in the `sendMessage` method:

```
//TODO Create a JMSProducer  
JMSProducer producer = context.createProducer();
```

- 4.2. Create a `TextMessage` using the `JMSPContext` interface to map the value of the `msg` parameter into the body of the JMS message:

```
//TODO Create a TextMessage  
TextMessage message = context.createTextMessage(msg);
```

- 4.3. Send the message to the destination, using the producer:

```
//TODO Send the message  
producer.send(helloworldQueue, message);
```

▶ **5.** Create a JMS consumer that reads a message from the `helloWorldQueue`.

- 5.1. Use the `createConsumer` method provided by the `JMSPContext` interface to build an instance of `MessageConsumer` for the `helloWorldQueue` destination in the `getMessage` method:

```
//TODO Create a JMS Consumer  
JMSPConsumer consumer = context.createConsumer(helloworldQueue);
```

- 5.2. Attempt to read a message from the queue, without waiting if there are no messages available. Use the `receiveNoWait` method provided by the `MessageConsumer` interface and cast the result into an instance `TextMessage`:

```
//TODO receive a message without waiting  
TextMessage msg = (TextMessage) consumer.receiveNoWait();
```

- 5.3. Close the consumer after everything is completed using the `close` method:

```
//TODO close the consumer  
consumer.close();
```

▶ **6.** Start EAP by selecting the **Servers** tab in the bottom pane of JBDS. Right-click the server **Red Hat JBoss EAP 7.0 [Stopped]**, and click the green button to start the server. Watch the **Console** until the server starts and displays the message:

```
INFO [org.jboss.as] (Controller Boot Thread) WFLYSRV0025: JBoss EAP 7.0.0.GA  
(WildFly Core 2.1.2.Final-redhat-1) started
```

▶ **7.** Deploy the application on JBoss EAP using Maven by running the following commands:

```
[student@workstation ~]$ cd /home/student/JB183/labs/jms-client
[student@workstation jms-client]$ mvn wildfly:deploy
```

Once complete, BUILD SUCCESS displays, as shown in the following example:

```
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 17.116 s
[INFO] Finished at: 2016-12-01T07:26:55-05:00
[INFO] Final Memory: 35M/210M
[INFO] -----
```

Validate the deployment in the server log shown in the **Console** tab in JBDS. When the app is deployed, the following appears in the log:

```
INFO [org.jboss.as.server] (management-handler-thread - 9) WFLYSRV0010: Deployed "jms-client.war" (runtime-name : "jms-client.war")
```

► **8.** Test the application in a browser.

- 8.1. Open <http://localhost:8080/jms-client> in a browser on the workstation VM.



Figure 8.3: Application home page

- 8.2. Attempt to read a message from the queue using the **Get Oldest Message On Queue** button. The **Queue is empty!** message is displayed.
- 8.3. Enter your name into the text field and press the **Submit** button. The app greets you and displays the current time on the server:

Enter your name:

• Hello TESTER!. Time on the server is: Oct 27 2017 01:01:31 PM

Figure 8.4: Application greeting the user by name with the current server time

- 8.4. Attempt to read a message from the queue using the **Get Oldest Message On Queue** button. This time, the last greeting is displayed:

Said Hello to TESTER at Oct 27 2017 01:01:31 PM

Figure 8.5: Getting the oldest message from the queue

► **9.** Undeploy the application and stop JBoss EAP.

- 9.1. Run the following command to undeploy the application:

```
[student@workstation jms-client]$ mvn wildfly:undeploy
```

- 9.2. To close the project, right-click the **jms-client** project in the **Project Explorer**, and select **Close Project**.
- 9.3. Right-click the Red Hat JBoss EAP 7.0 server in the **JBDS Servers** tab, and click **Stop**.

This concludes the guided exercise.

Creating MDBs

Objective

After completing this section, students should be able to create, package, and deploy a Message-driven bean.

Understanding the Life Cycle of a Message-driven Bean

The Java EE specification includes *message driven beans* (MDBs) as a mechanism to enable asynchronous consumption of messages from a JMS destination. Unlike other Java beans, MDBs do not expose public methods or functionality for other beans to access through dependency injection. Instead, all communication with an MDB happens over JMS. Each MDB is configured to listen on a specific JMS destination using an administered object.

The application server, JBoss EAP, is responsible for managing the life cycle of an MDB. The application server defines a pool of MDBs, which enables concurrent processing of messages. Concurrent message processing offers a substantial improvement in message throughput. The server creates the MDBs in the pool automatically on startup. When the destination that the MDB is listening on receives a new message, the application container automatically invokes the `onMessage()` method on one of the pre-created MDB instances. The `MessageListener` interface, which all MDBs must implement, requires this `onMessage()` method. Once the MDB is finished processing, the MDB instance is returned to the pool to be reused. Using a pool of MDBs improves application performance because when the destination receives messages, the MDB class is already instantiated and ready to process the message immediately.

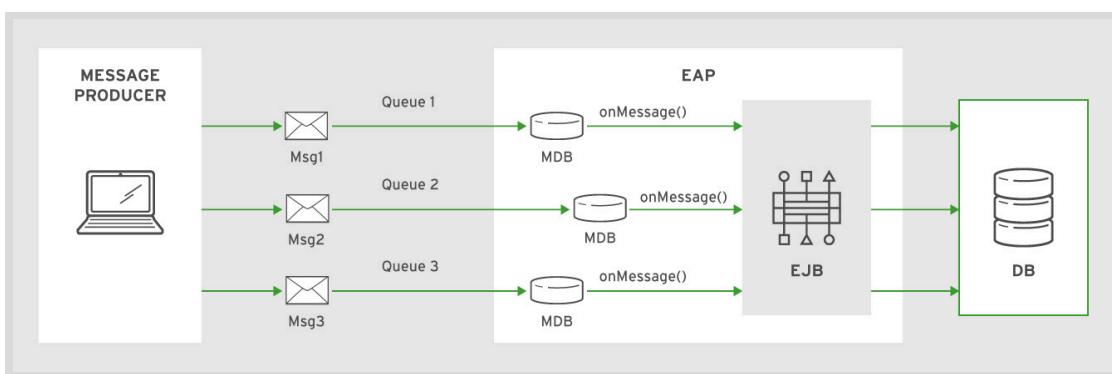


Figure 8.6: MDBs listen to specific queues

Because of the MDB's asynchronous execution and pooling behavior, the MDB's implementation has several requirements. MDBs must be completely stateless, retaining no conversational state or message data. However, MDBs can use injection to access other stateful resources, such as JMS provider connection information, a database connection, or an instance of another EJB. Additionally, all instances of an MDB class must be equal, and have no relationship with any specific client, enabling the application container to choose from them all, enabling concurrency.

The JMS `MessageConsumer` interface differs from an MDB in a number of important ways. The developer must manually invoke the `MessageConsumer`, whereas an MDB is automatically created. In addition, the JMS consumer only allows synchronous checking for new messages and is

typically single-threaded unless the developer manually implements concurrency. In contrast, the MDB is asynchronous and multi-threaded. For these reasons, MDBs are a more robust solution for Java EE applications that need to consume messages from a destination asynchronously.

Reviewing the Message Listener Interface

All MDBs must implement the `MessageListener` interface. The only public method required by this interface is the `onMessage` method, which takes a JMS message as an argument and has a `void` return type. The application server automatically calls this method and passes the JMS message that was received on the destination as the argument. The MDB developer is responsible for implementing the custom message processing logic that executes when `onMessage` is called.

The following `MessageListener` example shows an `onMessage` method that checks the message type and logs the message content:

```
public class QueueListener implements MessageListener {  
  
    private final static Logger LOGGER = Logger.getLogger(this.class.getName());  
  
    public void onMessage(Message rcvMessage) {  
        TextMessage msg = null;  
  
        try {  
            if (rcvMessage instanceof TextMessage) {  
                msg = (TextMessage) rcvMessage;  
                LOGGER.info("Received Message from helloworldQueue ===> " + msg.getText());  
            } else {  
                LOGGER.warn("Incorrect Message Type!");  
            }  
        } catch (JMSException e) {  
            throw new RuntimeException(e);  
        }  
    }  
    ...  
}
```

Notice that the example method is completely stateless, and retains no message data. The example also makes no assumptions about the message type, as this cannot be guaranteed. The MDB must be able to fail gracefully if it receives an unexpected message type.

Using Annotations to Configure an MDB

Finally, you activate the MDB, register it with the container, and configure the administered objects that the MDB uses to determine the destination to listen on. To activate an MDB and associate it with a destination, use the `@MessageDriven` annotation. This annotation uses *activation configuration properties* to configure the MDB.

MDBs require only one property, `destinationLookup`, which is the JNDI name of the destination administered object. MDBs also support some optional properties used to customize the MDB connection or behavior. Pass these properties into the `@MessageDriven` annotation using the `activationConfig` parameter. Use a `@ActivationConfigProperty` annotation for each property, specifying the property name with the `propertyName` attribute and the value of the property with the `propertyValue` attribute.

The following shows an example of these annotations configuring an MDB:

```

@MessageDriven(name = "QueueListener"①, activationConfig ② = {
    @ActivationConfigProperty(propertyName = "destinationLookup"③, propertyValue =
    "queue/helloworldQueue"),
    @ActivationConfigProperty(propertyName = "destinationType"④, propertyValue =
    "javax.jms.Queue")
})
public class QueueListener implements MessageListener {

```

- ① The name attribute sets the name of the MDB to `QueueListener`. This is the name that the MDB uses to register itself into the JNDI tree. This attribute is optional, and the name defaults to the class name if not otherwise specified.
- ② The `activationConfig` attribute passes in a set of `@ActivationConfigProperty` annotations to configure the MDB.
- ③ The `destinationLookup` property sets the JNDI name of the destination that the MDB should listen on.
- ④ The `destinationType` property specifies whether the destination is a queue or a topic. This property is optional.

The following table summarizes a few of the other available activation configuration properties:

JMS Activation Configuration Properties

Property Name	Property Description
<code>destinationLookup</code>	The JNDI name of the queue or topic. This property is mandatory.
<code>connectionFactoryLookup</code>	The JNDI name of the connection factory to use to connect to the destination. This property is optional, and if not specified the pooled connection factory named <code>activemq-ra</code> is used.
<code>destinationType</code>	The type of the destination, either <code>javax.jms.Queue</code> or <code>javax.jms.Topic</code> . This property is mandatory.
<code>messageSelector</code>	A string used to select a subset of the available messages. The syntax is similar to SQL syntax and is described in detail in JMS specification, and is beyond the scope of this course. This property is optional.
<code>acknowledgeMode</code>	The type of acknowledgment when not using transacted JMS. Valid values are <code>Auto-acknowledge</code> or <code>Dups-ok-acknowledge</code> . This property is optional. If not specified, the default value is <code>Auto-acknowledge</code> .



References

Further information is available in the JMS chapter of the *Configuring Messaging* guide for Red Hat JBoss EAP 7; at <https://access.redhat.com/documentation/en/red-hat-jboss-enterprise-application-platform/>

► Guided Exercise

Create a Message-driven Bean

In this exercise, you will create a message driven bean to read messages from a queue asynchronously.

Outcome

You should be able to implement a message driven bean to read messages from a queue.

Before You Begin

Open a terminal window on the **workstation** VM and run the following command to download the files required for this exercise.

```
[student@workstation ~]$ lab create-mdb setup
```

Steps

- ▶ 1. Open JBDS and import the Maven project.
 - 1.1. Open JBDS by double-clicking the JBoss Developer Studio icon on the workstation desktop. Set the workspace to /home/student/JB183/workspace and click OK.
 - 1.2. In the JBDS menu, click **File > Import** to open the **Import** wizard.
 - 1.3. On the **Select** page, click **Maven > Existing Maven Projects**, and then click **Next**.
 - 1.4. In the **Maven projects** page, click **Browse** to open the **Select root folder** window. Navigate to the /home/student/JB183/labs/ directory. Select the **create-mdb** folder, and then click **OK**.
 - 1.5. On the **Maven projects** page, click **Finish**.
 - 1.6. Watch the JBDS status bar to monitor the progress of the import operation. It may take a few minutes to download all the required dependencies.
- ▶ 2. Review the MDB pool and administered objects that are defined in the JBoss EAP configuration file.
 - 2.1. Using your preferred text editor, open the EAP configuration file at /opt/eap/standalone/configuration/standalone-full.xml.

```
[student@workstation ~]$ less /opt/eap/standalone/configuration/\\
standalone-full.xml
```

- 2.2. View the configuration of the MDB thread pool in the **standalone-full.xml** file by navigating to the **urn:jboss:domain:ejb3:4.0** subsystem:

```

<subsystem xmlns="urn:jboss:domain:ejb3:4.0">
  ...
  <mdb>
    <resource-adapter-ref resource-adapter-name="${ejb.resource-adapter-
name:activemq-ra.rar}"/>
    <bean-instance-pool-ref pool-name="mdb-strict-max-pool"/>①
  </mdb>
  <pools>
    <bean-instance-pools>
      <strict-max-pool name="slsb-strict-max-pool" derive-size="from-worker-pools"
instance-acquisition-timeout="5" instance-acquisition-timeout-unit="MINUTES"/>
      <strict-max-pool name="mdb-strict-max-pool" derive-size="from-cpu-count"
instance-acquisition-timeout="5" instance-acquisition-timeout-unit="MINUTES"/>②
    </bean-instance-pools>
  </pools>
  ...
</subsystem>
```

- ①** Configure the pool named `mdb-strict-max-pool` as the thread pool to be used by JBoss for all MDBs.
- ②** Create the thread pool named `mdb-strict-max-pool`, setting its size relative to the total CPU count, and defining a timeout value for how long to wait when there are no MDBs available before an error is thrown.

2.3. View the administered object created for the `helloworldQueue` queue by navigating to the `urn:jboss:domain:messaging-activemq:1.0` subsystem:

```

<subsystem xmlns="urn:jboss:domain:messaging-activemq:1.0">
  <server name="default">
    ...
    <jms-queue name="ExpiryQueue" entries="java:/jms/queue/ExpiryQueue"/>
    <jms-queue name="DLQ" entries="java:/jms/queue/DLQ"/>
    <jms-queue name="helloworldQueue" entries="queue/helloworldQueue java:jboss/
jms/queue/helloworldQueue"/>
    <jms-queue name="TodoListQueue" entries="queue/TodoListQueue java:jboss/jms/
queue/TodoListQueue"/>
    <connection-factory name="InVmConnectionFactory" entries="java:/
ConnectionFactory" connectors="in-vm"/>
    <connection-factory name="RemoteConnectionFactory" entries="java:jboss/
exported/jms/RemoteConnectionFactory" connectors="http-connector"/>
    <pooled-connection-factory name="activemq-ra" transaction="xa" entries="java:/
JmsXA java:jboss/DefaultJMSSConnectionFactory" connectors="in-vm"/>
  </server>
</subsystem>
```

Note that the `helloworldQueue` has a JNDI entry of `java:jboss/jms/queue/helloworldQueue`.

2.4. Close the text editor and **do not** save any changes to the `standalone-full.xml` file.

**Warning**

Problems can occur in the lab if unexpected changes are introduced in the configuration file.

- 3. Review the PersonService EJB class.

Open the PersonService class by expanding the **create-mdb** item in the Project Explorer tab in the left pane of JBDS, then click **create-mdb > Java Resources > src/main/java > com.redhat.training.service** and expand it. Double-click the PersonService.java file.

```
@Inject
JMSClient jmsUtil;
...
// Send a JMS message to the 'helloworldQueue'
jmsUtil.sendMessage("Said Hello to " + name.toUpperCase() + " at " + fdate);
...
```

Observe that this service class uses an instance of the **JMSClient** bean to send a message to a queue each time a person is greeted by the application.

- 4. Review the JMSClient EJB class.

Open the **JMSClient** class by expanding the **create-mdb** item in the Project Explorer tab in the left pane of JBDS, then click **create-mdb > Java Resources > src/main/java > com.redhat.training.messaging** and expand it. Double-click the **JMSClient.java** file.

```
@Resource(mappedName = "java:jboss/jms/queue/helloworldQueue")
private Queue helloworldQueue;
@Inject
JMSContext context;
...
JMSProducer producer = context.createProducer();
```

Observe that this class creates a **JMSProducer** and uses a **@Resource** annotation to access the administered object for the **helloworldQueue** using its JNDI entry.

- 5. Create the new message driven bean class.

5.1. Right-click **com.redhat.training.messaging** and click **New > Class**.

5.2. In the **Name** field, enter **QueueListener**. Click **Finish**.

5.3. Make the new class implement the **MessageListener** interface:

```
package com.redhat.training.messaging;

import javax.jms.MessageListener;

public class QueueListener implements MessageListener{
}
```

- 5.4. Resolve the compilation error by adding the `onMessage(Message message)` method required by the interface:

```
package com.redhat.training.messaging;

import javax.jms.MessageListener;
import javax.jms.Message;

public class QueueListener implements MessageListener{

    @Override
    public void onMessage(Message rcvMessage) {

    }

}
```

- 5.5. Implement the `onMessage` method to do a simple check of the message type and log the result to the server log:

```
package com.redhat.training.messaging;

import javax.jms.MessageListener;
import javax.jms.Message;
import javax.jms.JMSEException;
import javax.jms.TextMessage;

public class QueueListener implements MessageListener{

    public void onMessage(Message rcvMessage) {
        TextMessage msg = null;
        try {
            if (rcvMessage instanceof TextMessage) {
                msg = (TextMessage) rcvMessage;
                System.out.println("Received Message from helloWorldQueue ==> " +
msg.getText());
            } else {
                System.out.println("Message of wrong type: " +
rcvMessage.getClass().getName());
            }
        } catch (JMSEException e) {
            throw new RuntimeException(e);
        }
    }
}
```



Note

A try-catch block is required because the `msg.getText()` call can throw a `JMSEException` if the message is malformed. Checking the message's type helps mitigate issues with unexpected message types.

- 5.6. Configure the MDB using the `@MessageDriven` annotation, setting the necessary `@ActivationConfigProperty` instances to read from the `helloworldQueue`:

```
package com.redhat.training.messaging;

import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.jms.MessageListener;
import javax.jms.Message;
import javax.jms.JMSEException;
import javax.jms.TextMessage;

@MessageDriven(name = "QueueListener", activationConfig = {①
    @ActivationConfigProperty(propertyName = "destinationLookup", propertyValue =
"queue/helloworldQueue"), ②
    @ActivationConfigProperty(propertyName = "destinationType", propertyValue =
"javax.jms.Queue") ③
})
public class QueueListener implements MessageListener {
    ...
}
```

- ① Use the `@MessageDriven` annotation to register this class as an MDB.
- ② Set the `destinationLookup` property to the JNDI name of the administered object for the queue.
- ③ Set the `destinationType` property to `javax.jms.Queue` because the destination is a JMS queue and not a topic.

- 5.7. Save your changes using `Ctrl+S`.
- ▶ 6. Start EAP by selecting the **Servers** tab in the bottom pane of JBDS. Right-click the server **Red Hat JBoss EAP 7.0 [Stopped]**, and click the green button to start the server. Watch the **Console** until the server starts and displays the message:

```
INFO [org.jboss.as] (Controller Boot Thread) WFLYSRV0025: JBoss EAP 7.0.0.GA
(WildFly Core 2.1.2.Final-redhat-1) started
```

- ▶ 7. Deploy the application on JBoss EAP using Maven by running the following commands:

```
[student@workstation ~]$ cd /home/student/JB183/labs/create-mdb
[student@workstation create-mdb]$ mvn wildfly:deploy
```

Confirm successful deployment in the server log shown in the **Console** tab in JBDS. When the app is deployed, the following appears in the log:

```
INFO [org.jboss.as.server] (management-handler-thread - 9) WFLYSRV0010: Deployed
"create-mdb.war" (runtime-name : "create-mdb.war")
```

- ▶ 8. Test the application in a browser.

- 8.1. Open `http://localhost:8080/create-mdb` in a browser on the workstation VM.

- 8.2. Enter **Shadowman** in the text box labeled **Enter your name:** and click **Submit**

The page updates with the message *Hello SHADOWMAN!. Time on the server is: Nov 02 2017 03:55:03 PM*

- 8.3. Check the JBoss logs to see if the MDB picked up the message.

In JBDS, open the **Console** tab to view the server logs. If the MDB is working successfully, the following message appears:

```
...Received Message from helloworldQueue ==> Said Hello to SHADOWMAN at Nov 02  
2017 03:55:03 PM
```

► **9.** Undeploy the application and stop JBoss EAP.

- 9.1. Run the following command to undeploy the application:

```
[student@workstation create-mdb]$ mvn wildfly:undeploy
```

- 9.2. To close the project, right-click the **create-mdb** project in the **Project Explorer**, and select **Close Project**.

- 9.3. Right-click the Red Hat JBoss EAP 7.0 server in the JBDS **Servers** tab and click **Stop**. This concludes the guided exercise.

► Lab

Creating Messaging Applications with JMS

In this lab, you will use a message producer to send messages to a queue every time an item is updated in the To Do List Application.

Outcome

You should be able to build a JMS application that uses a JMS producer to put messages onto a queue and a message driven bean to listen on the same queue and log the messages to a special file.

Before You Begin

Open a terminal window on the `workstation` VM and run the following command to download the files required for this lab.

```
[student@workstation ~]$ lab messaging-lab setup
```

1. Open JBDS and import the `messaging-lab` project located in the `/home/student/JB183/labs/messaging-lab` directory.
2. Review the JNDI name assigned to the administered object defined for the `TodoListQueue` in the JBoss configuration file.
3. Create a new stateless EJB class named `JMSClient` that provides a public method called `sendMessage(String msg)` to send a message to the `TodoListQueue` using a JMS message producer.
4. Update the `JMSClient` EJB to inject the default `JMSContext`, also inject the administered object for the `TodoListQueue` and then use that context to create a `JMSProducer` to send a message to the queue.
 - 4.1. Inject the default `JMSContext` to access the default connection factory.
 - 4.2. Inject the `TodoListQueue` administered object with a `@Resource` annotation.
 - 4.3. Implement the `sendMessage(String msg)` method to place a new message on the queue using the `JMSProducer` interface, handling any exceptions by printing their stack trace to the console.
 - 4.4. Save your changes using `Ctrl+S`.
5. Update the `ItemService` to inject the `JMSClient` EJB. Add a call to the `update()` method in the `ItemService` class to use the injected `JMSClient` instance to send a JMS message every time an item is updated.
 - 5.1. Open the `ItemService` class by expanding the `messaging-lab` item in the `Project Explorer` tab in the left pane of JBDS, then click on `messaging-lab > Java Resources > src/main/java > com.redhat.training.todo.service` to expand it. Double-click the `ItemService.java` file.

- 5.2. Import the `JMSClient` and inject the default instance.
- 5.3. In the `update` method, add a call to the `sendMessage` method using the following message: `Item with ID:" + item.getId() + " was updated with status Done=" + item.isDone():`
- 5.4. Save your changes using `Ctrl+S`.
6. Update the `QueueListener` class to be a message driven bean that listens to the `TodoListQueue` and outputs messages to a special log file using the `writeMessageToFile` method.
 - 6.1. Open the `QueueListener` class by expanding the `messaging-lab` item in the **Project Explorer** tab in the left pane of JBDS, then click on `messaging-lab > Java Resources > src/main/java > com.redhat.training.todo.messaging` to expand it. Double-click the `QueueListener.java` file.
 - 6.2. Make the `QueueListener` class implement the `MessageListener` interface, and implement the `onMessage()` method to fix any compilation errors:
 - 6.3. Implement a simple check of the message type to ensure that it is an instance of `TextMessage` and log the result to the custom log file using the provided `writeMessageToFile(String message)` method. Be sure to use a try-catch block to handle any `JMSEExceptions`:
 - 6.4. Configure the MDB using the `@MessageDriven` annotation, setting the necessary `@ActivationConfigProperty` instances to read from the `TodoListQueue`:
 - 6.5. Save your changes using `Ctrl+S`.
7. Start JBoss EAP from within JBDS.
8. Build and deploy the application to JBoss EAP using Maven.
9. Test the application in a browser.
 - 9.1. Open Firefox on the `workstation` VM, and navigate to `http://localhost:8080/messaging-lab` to access the application.
 - 9.2. Add at least two new to-do items using the To Do List application interface.
 - 9.3. After the new items are successfully added, update the status of those items to done.
 - 9.4. Open a terminal window on the `workstation` VM and run the following command to navigate to the project directory and view the `ItemStatusLog.txt` log file:

```
[student@workstation ~]$ cd /home/student/JB183/labs/messaging-lab  
[student@workstation messaging-lab]$ cat ItemStatusLog.txt
```

A properly functioning MDB prints text similar to the following in the log:

```
Item with ID:11 was updated with status Done=true  
Item with ID:12 was updated with status Done=true
```

10. Open a new terminal window and run the following command to grade the lab:

```
[student@workstation ~]$ lab messaging-lab grade
```

The grading script should report SUCCESS. If there is a failure, check the errors and fix them until you see a SUCCESS message.

11. Clean up.

11.1. Undeploy the application from JBoss EAP using Maven with the following commands:

```
[student@workstation ~]$ cd /home/student/JB183/labs/messaging-lab  
[student@workstation messaging-lab]$ mvn wildfly:undeploy
```

11.2. Right-click on the **messaging-lab** project in the **Project Explorer**, and select **Close Project** to close this project.

11.3. Right-click on the Red Hat JBoss EAP 7.0 server in the JBDS **Servers** tab and click **Stop**.

This concludes the lab.

► Solution

Creating Messaging Applications with JMS

In this lab, you will use a message producer to send messages to a queue every time an item is updated in the To Do List Application.

Outcome

You should be able to build a JMS application that uses a JMS producer to put messages onto a queue and a message driven bean to listen on the same queue and log the messages to a special file.

Before You Begin

Open a terminal window on the **workstation** VM and run the following command to download the files required for this lab.

```
[student@workstation ~]$ lab messaging-lab setup
```

1. Open JBDS and import the **messaging-lab** project located in the **/home/student/JB183/labs/messaging-lab** directory.
 - 1.1. To open JBDS, double-click the JBoss Developer Studio icon on the workstation desktop. Leave the default workspace (**/home/student/JB183/workspace**) and click **OK**.
 - 1.2. In the JBDS menu, click **File > Import** to open the **Import** wizard.
 - 1.3. On the **Select** page, click **Maven > Existing Maven Projects**, and then click **Next**.
 - 1.4. In the **Maven projects** page, click **Browse** to open the **Select root folder** window. Navigate to the **/home/student/JB183/labs/** directory. Select the **messaging-lab** folder, and then click **OK**.
 - 1.5. On the **Maven projects** page, click **Finish**.
 - 1.6. Watch the JBDS status bar to monitor the progress of the import operation. It may take a few minutes to download all the required dependencies.
2. Review the JNDI name assigned to the administered object defined for the **TodoListQueue** in the JBoss configuration file.
 - 2.1. Using your preferred text editor, open the EAP configuration file at **/opt/eap/standalone/configuration/standalone-full.xml**:

```
[student@workstation ~]$ less /opt/eap/standalone/configuration/standalone-full.xml
```

- 2.2. Navigate to the urn:jboss:domain:messaging-activemq:1.0 subsystem:

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:1.0">
<server name="default">
  ...
  <jms-queue name="ExpiryQueue" entries="java:/jms/queue/ExpiryQueue"/>
  <jms-queue name="DLQ" entries="java:/jms/queue/DLQ"/>
  <jms-queue name="helloworldQueue" entries="queue/helloworldQueue java:jboss/jms/
queue/helloworldQueue"/>
  <jms-queue name="TodoListQueue" entries="queue/TodoListQueue java:jboss/jms/
queue/TodoListQueue"/>
    <connection-factory name="InVmConnectionFactory" entries="java:/
ConnectionFactory" connectors="in-vm"/>
    <connection-factory name="RemoteConnectionFactory" entries="java:jboss/exported/
jms/RemoteConnectionFactory" connectors="http-connector"/>
    <pooled-connection-factory name="activemq-ra" transaction="xa" entries="java:/
JmsXA java:jboss/DefaultJMSCConnectionFactory" connectors="in-vm"/>
  </server>
</subsystem>
```

Note that the TodoListQueue has a JNDI entry of `java:jboss/jms/queue/TodoListQueue`.

- 2.3. Close the text editor and **do not** save any changes to the `standalone-full.xml` file.



Warning

Problems can occur in the lab if unexpected changes are introduced in the configuration file.

3. Create a new stateless EJB class named `JMSClient` that provides a public method called `sendMessage(String msg)` to send a message to the `TodoListQueue` using a JMS message producer.
 - 3.1. In the **Project Explorer** tab in the left pane of JBDS, then click on `messaging-lab > Java Resources > src/main/java > com.redhat.training.todo.messaging`, right-click `com.redhat.training.messaging` and select **New > Class**.
 - 3.2. In the **Name** field, enter `JMSClient`. Click **Finish**.
 - 3.3. Edit the newly created `JMSClient` class, add the `@Stateless` annotation to mark it as an EJB available for injection.

```
import javax.ejb.Stateless;

@Stateless
public class JMSClient {
```

- 3.4. Save your changes using **Ctrl+S**.
4. Update the `JMSClient` EJB to inject the default `JMSContext`, also inject the administered object for the `TodoListQueue` and then use that context to create a `JMSPublisher` to send a message to the queue.

- 4.1. Inject the default `JMSContext` to access the default connection factory.

```
import javax.ejb.Stateless;
import javax.inject.Inject;
import javax.jms.JMSContext;

@Stateless
public class JMSClient {
    @Inject
    JMSContext context;
```

- 4.2. Inject the `TodoListQueue` administered object with a `@Resource` annotation.

```
import javax.ejb.Stateless;
import javax.inject.Inject;
import javax.jms.JMSContext;
import javax.jms.Queue;
import javax.annotation.Resource;

@Stateless
public class JMSClient {

    @Inject
    JMSContext context;

    @Resource(mappedName = "java:jboss/jms/queue/TodoListQueue")
    private Queue todoListQueue;
```

- 4.3. Implement the `sendMessage(String msg)` method to place a new message on the queue using the `JMSProducer` interface, handling any exceptions by printing their stack trace to the console.

```
import javax.ejb.Stateless;
import javax.inject.Inject;
import javax.jms.JMSContext;
import javax.jms.Queue;
import javax.annotation.Resource;
import javax.jms.JMSProducer;
import javax.jms.TextMessage;

@Stateless
public class JMSClient {

    @Inject
    JMSContext context;

    @Resource(mappedName = "java:jboss/jms/queue/TodoListQueue")
    private Queue todoListQueue;

    public void sendMessage(String msg) {
        try {
            JMSProducer producer = context.createProducer();
            TextMessage message = context.createTextMessage(msg);
```

```

producer.send(todoListQueue, message);
System.out.println("Sent Message: " + msg);
}
catch (Exception e) {
e.printStackTrace();
}
}
...

```

4.4. Save your changes using **Ctrl+S**.

5. Update the `ItemService` to inject the `JMSClient` EJB. Add a call to the `update()` method in the `ItemService` class to use the injected `JMSClient` instance to send a JMS message every time an item is updated.
 - 5.1. Open the `ItemService` class by expanding the `messaging-lab` item in the Project Explorer tab in the left pane of JBDS, then click on `messaging-lab > Java Resources > src/main/java > com.redhat.training.todo.service` to expand it. Double-click the `ItemService.java` file.
 - 5.2. Import the `JMSClient` and inject the default instance.

```

...
import com.redhat.training.todo.messaging.JMSClient;

@Stateless
public class ItemService {

    @Inject
    private JMSClient jmsClient;

```

5.3. In the `update` method, add a call to the `sendMessage` method using the following message: `Item with ID:" + item.getId() + " was updated with status Done=" + item.isDone()`:

```

public void update(Item item) {
    jmsClient.sendMessage("Item with ID:" + item.getId() + " was updated with status
    Done=" + item.isDone());
    em.merge(item);
}

```

5.4. Save your changes using **Ctrl+S**.

6. Update the `QueueListener` class to be a message driven bean that listens to the `TodoListQueue` and outputs messages to a special log file using the `writeMessageToFile` method.
 - 6.1. Open the `QueueListener` class by expanding the `messaging-lab` item in the Project Explorer tab in the left pane of JBDS, then click on `messaging-lab > Java Resources > src/main/java > com.redhat.training.todo.messaging` to expand it. Double-click the `QueueListener.java` file.
 - 6.2. Make the `QueueListener` class implement the `MessageListener` interface, and implement the `onMessage()` method to fix any compilation errors:

```
package com.redhat.training.messaging;

import javax.jms.MessageListener;

public class QueueListener implements MessageListener{
    @Override
    public void onMessage(Message rcvMessage) {

    }
}
```

- 6.3. Implement a simple check of the message type to ensure that it is an instance of `TextMessage` and log the result to the custom log file using the provided `writeMessageToFile(String message)` method. Be sure to use a try-catch block to handle any `JMSExceptions`:

```
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;

public class QueueListener implements MessageListener{

    public void onMessage(Message rcvMessage) {
        TextMessage msg = null;
        try {
            if (rcvMessage instanceof TextMessage) {
                msg = (TextMessage) rcvMessage;
                System.out.println("Received Message from TodoListQueue ===> " +
msg.getText());
                writeMessageToFile(msg.getText());
            } else {
                System.out.println("Message of wrong type: " +
rcvMessage.getClass().getName());
            }
        } catch (JMSException e) {
            e.printStackTrace();
        }
    }
}
```

- 6.4. Configure the MDB using the `@MessageDriven` annotation, setting the necessary `@ActivationConfigProperty` instances to read from the `TodoListQueue`:

```
import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;

@MessageDriven(name = "QueueListener", activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationLookup", propertyValue =
"queue/TodoListQueue"),
    @ActivationConfigProperty(propertyName = "destinationType", propertyValue =
"javax.jms.Queue")}
```

```
}
public class QueueListener implements MessageListener {
...
}
```

- 6.5. Save your changes using **Ctrl+S**.
 7. Start JBoss EAP from within JBDS.
Select the **Servers** tab in JBDS. Right-click on the server entry **Red Hat JBoss EAP 7.0 [Stopped]** and click the green Start option to start the server. Watch the **Console** tab of JBDS until the server starts and you see the message:
- ```
INFO [org.jboss.as] (Controller Boot Thread) WFLYSRV0025: JBoss EAP 7.0.2.GA
(WildFly Core 2.1.8.Final-redhat-1) started
```
8. Build and deploy the application to JBoss EAP using Maven.  
Open a new terminal, and change directory to the `/home/student/JB183/labs/messaging-lab` folder.
- ```
[student@workstation ~]$ cd /home/student/JB183/labs/messaging-lab
```
- Build and deploy the EJB to JBoss EAP by running the following command:
- ```
[student@workstation messaging-lab]$ mvn clean wildfly:deploy
```
- A successful build displays the following output:
- ```
[student@workstation messaging-lab]$ mvn clean package wildfly:deploy
...
[INFO] [INFO] <<< wildfly-maven-plugin:1.0.2.Final:deploy (default-cli) < package
@ messaging-lab <<<
...
[INFO] --- maven-war-plugin:2.1.1:war (default-war) @ messaging-lab ---
...
[INFO] --- wildfly-maven-plugin:1.0.2.Final:deploy (default-cli) @ messaging-lab
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
```
9. Test the application in a browser.
- 9.1. Open Firefox on the **workstation** VM, and navigate to `http://localhost:8080/messaging-lab` to access the application.
 - 9.2. Add at least two new to-do items using the To Do List application interface.
 - 9.3. After the new items are successfully added, update the status of those items to done.
 - 9.4. Open a terminal window on the **workstation** VM and run the following command to navigate to the project directory and view the `ItemStatusLog.txt` log file:

```
[student@workstation ~]$ cd /home/student/JB183/labs/messaging-lab
[student@workstation messaging-lab]$ cat ItemStatusLog.txt
```

A properly functioning MDB prints text similar to the following in the log:

```
Item with ID:11 was updated with status Done=true  
Item with ID:12 was updated with status Done=true
```

10. Open a new terminal window and run the following command to grade the lab:

```
[student@workstation ~]$ lab messaging-lab grade
```

The grading script should report SUCCESS. If there is a failure, check the errors and fix them until you see a SUCCESS message.

11. Clean up.

- 11.1. Undeploy the application from JBoss EAP using Maven with the following commands:

```
[student@workstation ~]$ cd /home/student/JB183/labs/messaging-lab  
[student@workstation messaging-lab]$ mvn wildfly:undeploy
```

- 11.2. Right-click on the **messaging-lab** project in the **Project Explorer**, and select **Close Project** to close this project.

- 11.3. Right-click on the Red Hat JBoss EAP 7.0 server in the JBDS **Servers** tab and click **Stop**.

This concludes the lab.

Summary

In this chapter, you learned:

- By enabling asynchronous processing between loosely-coupled systems, messaging allows developers to build applications that are more efficient, more easily scaled, and more reliable.
- Point-to-point messaging is when two applications are connected using a queue, and each message that a producer sends is received by a single consumer.
- Publish-subscribe messaging is when multiple applications subscribe to a topic, and any message sent to that topic is duplicated and copied to each of the subscribers.
- A common practice when working with messaging on an application server such as JBoss EAP 7 is to maintain JMS destinations and connection factories administratively in the server configuration rather than in the application code itself.
- There are three parts to the basic structure of a JMS message: headers, properties, and a body.
- The `@Resource` annotation directly injects a destination object, such as a `Queue` or `Topic`, as well as a `ConnectionFactory` object using only the JNDI name defined in the server configuration.
- A message producer is a type of JMS object created by a `JMSContext` object that sends JMS messages to a destination.
- A message consumer is a type of JMS object created by a `JMSContext` object that receives JMS messages from a destination synchronously using a method call.
- Message-driven beans (MDBs) enable asynchronous consumption of messages from a JMS destination.

Chapter 9

Securing Java EE Applications

Overview

Goal Secure a Java EE application with JAAS.

Objectives

- Describe the JAAS specification.
- Configure a security domain in the JBoss EAP application server.
- Secure a REST API with authentication and role-based authorization.

Sections

- Describing the JAAS Specification (and Quiz)
- Configuring a Security Domain in JBoss EAP (and Guided Exercise)
- Securing a REST API (and Guided Exercise)

Lab Securing Java EE Applications

Describing the JAAS Specification

Objective

After completing this section, students should be able to describe the JAAS specification.

Java EE Security

When building enterprise applications that access sensitive data, security must be a top priority for developers. Developers need to have a clear understanding of who is able to access the application and which parts of the application those users are allowed to use. Defining which users have access to an application is known as *authentication*, while defining the permissions within the application for those users is known as *authorization*. Ideally, when defining access restrictions to various application components, users are limited only to the minimal amount of access that each user requires. To customize authorization in an application, restrictions are applied to a *user*, which represents an individual, or to a *role*, which refers to a defined group of users.

For example, consider an online bookstore web application where customers purchase books online, and the store owner manages the inventory. The user `shadowman` is a customer accessing the site and has the role `customer`. The site administrator with the user name `redhat` has the role `admin`. The server authenticates the users `shadowman` and `redhat` to ensure that each user matches its password. Once authenticated, the EJB methods are annotated to restrict access to individual user roles. Because customers are not allowed to manage the inventory of the store, the users with the role `customer` are not able to invoke methods that manage the inventory, while the users with the role `admin` can make inventory changes.

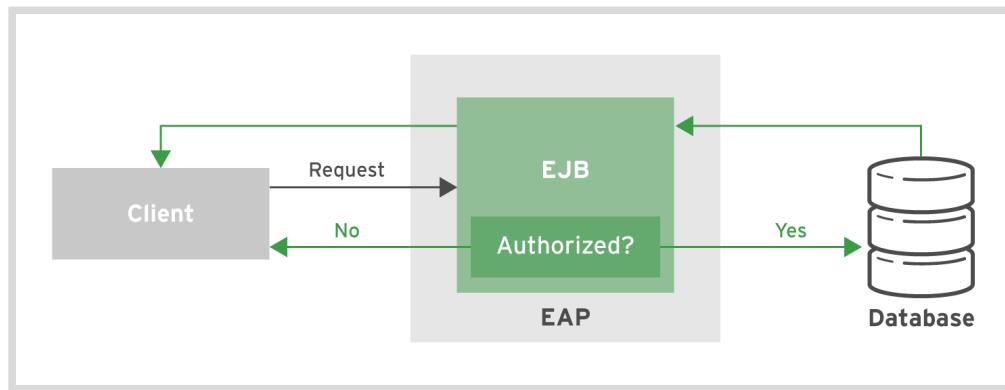


Figure 9.1: Secure EJB authorization

Java Authentication and Authorization Service (JAAS) is a security API that is used to implement user authentication and authorization in Java applications (JSR-196). The API extends the Java Enterprise Edition access control architecture to support user-based authorization. Java EE enforces the API as a mechanism to enable and guarantee access. JAAS provides declarative role-based security in the JBoss Enterprise Application Platform. *Declarative security* separates security concerns from application code by using the container to manage security. The container provides an authorization system based on annotations and XML descriptors within the application code that secures resources. This approach is in contrast to *programmatic security*, which requires each application to contain code that manages security.

Declarative Security

Using declarative security requires developers and administrators to leverage annotations and deployment descriptors to define the security behavior of an application. An EJB, for example, can restrict aspects of the application based on a user's role using only annotations. It does not require an application to manage the security context. To manage aspects of security such as managing authentication and authorization, you need deployment descriptors, responsible for dictating how an application server deploys the application and how the server secures the application. This is set in the application's `web.xml` or, when developing with Red Hat JBoss EAP, in `jboss-web.xml`.

Developers use the `web.xml` file to define which resources in an application should be secured, how they are secured, and what data is used to validate the credentials. The following is a snippet of the `web.xml` that defines BASIC authentication:

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>All resources</web-resource-name>
        <url-pattern>/*</url-pattern>①
    </web-resource-collection>
    <auth-constraint>
        <role-name>*</role-name>②
    </auth-constraint>
</security-constraint>
<login-config>
    <auth-method>BASIC</auth-method>③
    <realm-name>basicRealm</realm-name>④
</login-config>
```

- ①** The resources that the security constraint applies to. The "/" indicates that all resources are secured.
- ②** The roles that are authorized to access the resources. In this instance, all roles may access the application.
- ③** The method the application uses to access the credentials of a user. BASIC prompts the user in a pop-up window as soon as the application is accessed.
- ④** The name of the realm that stores the user credential information. This topic is discussed further in the next section.

The `jboss-web.xml` file adds additional JBoss-specific descriptions, such as how the server handles authentication and authorization for the application. In many cases, this file is used to define a security domain, which is a set of JAAS declarative security configurations. This file and other JBoss-specific security configurations are discussed in the next section.

Using deployment descriptors to define aspects of security can be helpful, but they are also severely limiting, especially in any application that has more than the most basic of security requirements. Annotations that are placed directly in the EJB application code provide a more flexible and customizable approach to security. This approach is useful for securing methods of REST APIs or for limiting certain roles to using only certain method calls within an application. Annotations that can be used to secure an EJB:

- `@SecurityDomain`: Located at the beginning of the class, this annotation defines the security domain by name to use for the EJB.

- **@DeclareRoles**: Located at the beginning of the class, this annotation defines the roles that are tested for permissions in the class. If this annotation is not used, the roles are checked based on the presence of the **@RolesAllowed** annotations.
- **@RolesAllowed**: Located either at the beginning of the class or before a method header, this annotation defines a list of one or more roles allowed to access a method. If placed before the class header, methods in the class without an annotation default to this annotation.
- **@PermitAll**: Located either at the beginning of the class or before a method header, this annotation specifies that all roles are allowed to access a method.
- **@DenyAll**: Located either at the beginning of the class or before a method header, this annotation specifies that no roles are allowed to access a method.
- **@RunAs**: Located either at the beginning of the class or before a method header, this annotation specifies the role used when running a method. This annotation is useful when an EJB is calling another EJB and needs to take on a new role for security restrictions in another EJB.

The following is an example of an EJB class that uses the `other` security domain to restrict authorization access to its methods:

```
@Stateless  
@RolesAllowed({"admin, qa"}) ①  
@SecurityDomain("other") ②  
public class HelloWorldEJB implements HW {  
    @PermitAll ③  
    public String HelloWorld(String msg) {  
        return "Hello " + msg;  
    }  
    @RolesAllowed("admin") ④  
    public String GoodbyeAdmin(String msg) {  
        return "See you later, " + msg;  
    }  
    ⑤  
    public String GoodbyeSecure(String msg) {  
        return "Adios, " + msg;  
    }  
}
```

- ➊ The `HelloWorldEJB` class defaults to restricting all of its methods to only being available to the `admin` and `qa` users.
- ➋ The class is leveraging the `other` security domain. In this case, this security domain utilizes the property files that store the role information.
- ➌ The `HelloWorld` method is available to all roles, not just `admin` and `qa`.
- ➍ The `GoodbyeAdmin` method is available only to users authenticated as the role `admin`.
- ➎ The `GoodbyeSecure` method is available to only `admin` and `qa` as the method defaults to the `RolesAllowed` defined at the class level.

Programmatic Security

In some cases, declarative security is not enough to meet all security requirements for an application. In these instances, developers may prefer also using programmatic security to

have more control over authentication and authorization decisions within the application. The `EJBContext` object contains information about the current security context and provides the following two useful methods for accessing information about the authenticated user that can be used to make authorization decisions:

- `isCallerInRole(String role)`: Returns a boolean indicating whether a user belong to a given **role**.
- `getCallerPrincipal()`: Returns the currently authenticated user.

The following example demonstrates using the `EJBContext` to identify a user and the user's role:

```
@Stateless  
public class HelloWorldEJB implements HW {  
    @Resource  
    EJBContext context;  
  
    public String HelloWorld() {  
        if (context.isCallerInRole("admin")) {  
            return "Hello " + context.getCallerPrincipal().getName();  
        } else {  
            return "Unauthorized user.";  
        }  
    }  
}
```

In this example, the `HelloWorld()` method uses the `EJBContext` to check to see if the user calling the method belongs to the role `admin`. If the user does belong to this role, a response is returned with the authenticated user's user name.

In addition to utilizing the `EJBContext`, the `HttpServletRequest` interface provides methods for programatically managing user authentication. The following methods are available for authenticating users with the `HttpServletRequest` interface:

- `authenticate(HttpServletRequest)`: Prompt the user to provide credentials for authentication.
- `login(String username, String password)`: Log in the user by providing the **username** and **password**.
- `logout()`: Log out the currently authenticated user.



References

JSR-196 Java Authentication Service Provider Interface for Containers

<https://www.jcp.org/en/jsr/detail?id=196>



References

Further information is available in the Security chapter of the *Development Guide* for Red Hat JBoss EAP 7 at
<https://access.redhat.com/documentation/en/red-hat-jboss-enterprise-application-platform/>

► Quiz

Describing the JAAS Specification

Match the items below to their counterparts in the table.

- A set of JAAS declarative security configurations in EAP.
- Defines the type of <login-config>, such as BASIC.
- Specifies the security configuration to be used in the application.
- The annotation defining which roles can access a method.
- The annotation defining which roles to check permissions for.
- The annotation that defining which role executes a method.
- Verification that a user has the privilege to do a certain action.
- Verification that a user is who they claim to be.

Term	Definition
Security Domain	
Authentication	
Authorization	
web.xml	
jboss-web.xml	
@DeclareRoles	
@RolesAllowed	
@RunAs	

► Solution

Describing the JAAS Specification

Match the items below to their counterparts in the table.

Term	Definition
Security Domain	A set of JAAS declarative security configurations in EAP.
Authentication	Verification that a user is who they claim to be.
Authorization	Verification that a user has the privilege to do a certain action.
web.xml	Defines the type of <login-config>, such as BASIC.
jboss-web.xml	Specifies the security configuration to be used in the application.
@DeclareRoles	The annotation defining which roles to check permissions for.
@RolesAllowed	The annotation defining which roles can access a method.
@RunAs	The annotation that defines which role executes a method.

Configuring a Security Domain in JBoss EAP

Objective

After completing this section, students should be able to configure a security domain in EAP.

Security Realms in EAP

Using an application server, such as Red Hat JBoss EAP, simplifies the configuration and implementation of security for developers and application administrators. EAP and other application servers provide utilities and predefined default configurations that help manage authentication and authorization. EAP manages the user security information in *security realms*. By default, EAP defines the ApplicationRealm, which uses the following files to store the users and their roles, respectively:

- application-users.properties
- application-roles.properties

Although more security realms can be added to EAP, this course primarily uses the ApplicationRealm for authentication and authorization. The following is the default configuration for the ApplicationRealm:

```
<security-realm name="ApplicationRealm">
    <authentication>
        <local default-user="$local" allowed-users="*" skip-group-
loading="true"/>
        <properties path="application-users.properties" relative-
to="jboss.server.config.dir"/>
    </authentication>
    <authorization>
        <properties path="application-roles.properties" relative-
to="jboss.server.config.dir"/>
    </authorization>
</security-realm>
```

Notice that the realm has a tag each for `<authentication>` and `<authorization>`. The `<authentication>` tag defines the path to the user properties file. In this case, that file is `application-users.properties` in the EAP server configuration directory. This file stores user names and passwords as a key-value pair, for example:

```
<username>=<password>
```



Note

The `EAP_HOME/bin/add-user.sh` script is a utility script that updates the default `application-users.properties` file with new users.

The <authorization> tag defines the path to the roles properties file. In this case, that file is `application-roles.properties`, which is located in the EAP server configuration directory. This file stores users and roles as key-value pairs using the following syntax:

```
<role>=<user1>,<user2>...
```

Login Modules

While the security realms in EAP are used to configure the user credentials, a `security-domain` defines a set of JAAS declarative security configurations. The biggest advantage of using a declarative approach to security is that it separates a lot of the security details from the application itself. This provides additional flexibility and keeps the application's business logic code readable and easier to maintain by removing the implementation details for the security technology that the application server uses.

EAP includes several built-in *login modules* that developers can use for authentication within a security domain. These login modules include the ability to read user information from a relational database, an LDAP server, or flat files. It is also possible to build a custom module depending on the security requirements of the application.

The method for user authentication is defined in the security domain. By default, EAP defines the `other` security domain with the following configuration:

```
<security-domain name="other" cache-type="default">
    <authentication>
        ...
        <login-module code="RealmDirect" flag="required">
            <module-option name="password-stacking" value="useFirstPass"/>
        </login-module>
    </authentication>
</security-domain>
```

The `RealmDirect` login module uses an application realm when making authentication and authorization decisions. If no realm is specified, then the module uses the `ApplicationRealm`, and therefore the users and roles property files for authentication and authorization.

To enable the `other` security domain in the application, add the following tag to the application's `src/main/webapp/WEB-INF/jboss-web.xml`:

```
<security-domain>other</security-domain>
```

Finally, to complete authentication, configure the `WEB-INF/web.xml` file in the application to define the <login-config>. The following example defines the <login-config> to use **BASIC** authentication with the `ApplicationRealm`. With this configuration in place, users are prompted for credentials when they access a server resource and the server verifies the credentials against the `ApplicationRealm`.

```
<login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>ApplicationRealm</realm-name>
</login-config>
</web-app>
```

UsersRoles Login Module

The UsersRoles login module is a simple module that is useful for testing some of the basic security functionality of an application. The module provides a way for developers to quickly authenticate users and verify that authorization restrictions are correctly configured. Similar to relying on the `ApplicationRealm` as the other security domain does, the `UsersRoles` module uses a properties files to store both user credentials and role data.

The following is an example of the `UsersRoles` login module:

```
<security-domain name="loginTest" ① cache-type="default">
    <authentication>
        ...
        <login-module code="UsersRoles" ② flag="required" ③>
            <module-option name="usersProperties" value="users.properties" ④/>
            <module-option name="rolesProperties" value="roles.properties" ⑤/>
        </login-module>
    </authentication>
</security-domain>
```

- ① The name of the security domain. This name is referenced in the `jboss-web.xml` file.
- ② The code to define which login module is being used. In this case, the `UsersRoles` login module is being configured.
- ③ The flag to define the behavior of the login module. `required` indicates that the module is required authentication to succeed.
- ④ The property to define the file name that stores all users and passwords as a key-value pair.
- ⑤ The property to define the file name that stores all users roles as a key-value pair.

Database Login Module

In a production environment, it is extremely uncommon to see user credentials and role information stored in locally-stored properties files. These modules and techniques are primarily used for testing purposes. One of the more practical solutions than local property files for managing user credentials is storing the information in a database. There are many benefits of using a database instead of a file to store user information. Databases can be easily shared across multiple application servers, they include robust data security and backup solutions, and they are efficient with a large data set. If an application uses the `Database` login module, the application users are stored in a database along with the roles that users are associated with.

The following is an example of the `Database` login module:

```
<security-domain name="db" cache-type="default">
    <authentication>
        ...
        <login-module code="Database" ① flag="required">
            <module-option name="dsJndiName" value="java:/DataSource" ②/>
                <module-option name="principalsQuery" value="select password from
Users where username=?" ③/>
                <module-option name="rolesQuery" value="select role, 'Roles' from UserRoles
where username=?" ④/>
```

```
</login-module>  
</authentication>  
</security-domain>
```

- ➊ The code to define which login module is used. In this case, the Database login module is being configured.
- ➋ The property to define the JNDI name used to access the datasource. Note that this datasource must already be configured.
- ➌ The property to define the query used to get the password for a given user. This query depends on how the database is configured.
- ➍ The property to define the query used to get the role for a given user. This query depends on how the database is configured.



References

Further information is available in the *Login Module Reference Guide* for Red Hat JBoss EAP 7 at
<https://access.redhat.com/documentation/en/red-hat-jboss-enterprise-application-platform/>



References

Further information is available in the Security chapter of the *Development Guide* for Red Hat JBoss EAP 7 at
<https://access.redhat.com/documentation/en/red-hat-jboss-enterprise-application-platform/>

► Guided Exercise

Configuring a Security Domain in JBoss EAP

In this exercise, you will secure a JEE application by creating a security domain.

Outcome

You should be able to use a security domain in EAP to secure authentication and authorization in a JEE application.

Before You Begin

Open a terminal window on `workstation` and run the following command to download the files required for this lab.

```
[student@workstation ~]$ lab security-domain setup
```

Steps

- ▶ 1. Open JBDS and import the Maven project.
 - 1.1. Open JBDS by double-clicking the JBoss Developer Studio icon on the workstation desktop. Set the workspace to `/home/student/JB183/workspace` and click **OK**.
 - 1.2. In the JBDS menu, click **File > Import** to open the **Import** wizard.
 - 1.3. On the **Select** page, click **Maven > Existing Maven Projects**, and then click **Next**.
 - 1.4. In the **Maven projects** page, click **Browse** to open the **Select root folder** window. Navigate to the `/home/student/JB183/labs/` directory. Select the **security-domain** folder, and then click **OK**.
 - 1.5. On the **Maven projects** page, click **Finish**.
 - 1.6. Watch the JBDS status bar to monitor the progress of the import operation. It may take a few minutes to download all the required dependencies.
- ▶ 2. Start EAP by selecting the **Servers** tab in the bottom pane of JBDS. Right-click the server **Red Hat JBoss EAP 7.0 [Stopped]**, and click the green button to start the server. Watch the **Console** until the server starts and displays the message:

```
INFO [org.jboss.as] (Controller Boot Thread) WFLYSRV0025: JBoss EAP 7.0.0.GA
(WildFly Core 2.1.2.Final-redhat-1) started
```

- ▶ 3. Create the **UsersRoles** security domain in JBoss EAP and verify that it utilizes the **hello-users.properties** and **hello-roles.properties** files for authentication and authorization.

- 3.1. In the terminal window, run the following script to create the security domain in the running EAP server:

```
[student@workstation ~]$ cd JB183/labs/security-domain  
[student@workstation security-domain]$ ./create-sd.sh
```

- 3.2. Using a text editor, open the /opt/jboss-eap-7.0/standalone/configuration/standalone-full.xml configuration file and observe the new security domain that directs the application server to use the users and roles property files in the project directory.

```
<security-domain name="userroles" cache-type="default">  
  <authentication>  
    <login-module code="UsersRoles" flag="required">  
      <module-option name="usersProperties" value="file:///home/student/JB183/labs/  
security-domain/hello-users.properties"/>  
      <module-option name="rolesProperties" value="file:///home/student/JB183/labs/  
security-domain/hello-users.properties"/>  
    </login-module>  
  </authentication>  
</security-domain>
```

- 4. Update the jboss-web.xml file to use the new security domain.

- 4.1. Open the jboss-web.xml file by expanding the **security-domain** item in the **Project Explorer** tab in the left pane of JBDS, then click on **security-domain > src/main/webapp > WEB-INF** and expand it. Double-click the jboss-web.xml file.
- 4.2. Update the jboss-web.xml file to use the new security domain named userroles:

```
<?xml version="1.0" encoding="ISO-8859-1"?>  
  
<jboss-web>  
  <security-domain>userroles</security-domain>  
</jboss-web>
```

- 4.3. Save your changes by pressing **Ctrl+S**.

- 5. Update the web.xml file to use BASIC authentication and restrict access to the application's admin.jsf.

- 5.1. Open the web.xml file by expanding the **security-domain** item in the **Project Explorer** tab in the left pane of JBDS, then click on **security-domain > src/main/webapp > WEB-INF** and expand it. Double-click the web.xml file.
- 5.2. The first security constraint refers to the index.html. This is the main page of the web application. Add the following <auth-constraint> to the index.html security constraint to restrict access to this resource to only users with the role guest and admin.

```
<security-constraint>  
  <web-resource-collection>  
    <web-resource-name>Index</web-resource-name>
```

```

<url-pattern>/index.html</url-pattern>
</web-resource-collection>
<!-- Add auth constraint here -->
<auth-constraint>
  <role-name>guest</role-name>
  <role-name>admin</role-name>
</auth-constraint>
</security-constraint>
```

- 5.3. Update the second security constraint to restrict access to the `admin.jsf` page to only users with the role `admin`. Add a new `auth-constraint` and update the `url-pattern`.

```

<security-constraint>
  <web-resource-collection>
    <web-resource-name>Secure resources</web-resource-name>
    <!-- Update url pattern -->
    <url-pattern>/admin.jsf</url-pattern>
  </web-resource-collection>
  <!-- Add auth constraint -->
  <auth-constraint>
    <role-name>admin</role-name>
  </auth-constraint>
</security-constraint>
```

- 5.4. Set the `auth-method` of the `login-config` to **BASIC** authentication:

```

<login-config>
  <!-- Update the auth method -->
  <auth-method>BASIC</auth-method>
</login-config>
```

- 5.5. Save your changes by pressing **Ctrl+S**.

- 6. Create a user with the role `guest` and a user with the role `admin` in the `hello-users.properties` and `hello-roles.properties` to access the web application.

- 6.1. In a new terminal window, navigate to `/home/student/JB183/labs/security-domain` directory and create a new file named `hello-users.properties`:

```
[student@workstation ~]$ cd /home/student/JB183/labs/security-domain
[student@workstation security-domain]$ gedit hello-users.properties
```

- 6.2. Update the file to contain the following content to create two new users, `customer` and `owner`, both with the password `redhat1!`:

```
customer=redhat1!
owner=redhat1!
```

- 6.3. Save the file and close the editor.

- 6.4. Create a file named `hello-roles.properties` in the `/home/student/JB183/labs/security-domain` directory to associate the new users to the roles `admin` and `guest`:

```
[student@workstation security-domain]$ gedit hello-roles.properties
```

- 6.5. Update the file to contain the following content to associate the role `guest` with the user `customer`, and the role `admin` with the user `owner`:

```
customer=guest
owner=admin
```

- 6.6. Save the file and close the editor.

- 7. Deploy the security-domain application using the following commands in the terminal window:

```
[student@workstation bin]$ cd /home/student/JB183/labs/security-domain
[student@workstation security-domain]$ mvn wildfly:deploy
```

- 8. Test the security-domain as both the `customer` and `owner` users by navigating to `http://localhost:8080/security-domain`.

- 8.1. In Firefox on the workstation VM, navigate to `http://localhost:8080/security-domain`.
- 8.2. When prompted, enter the following credentials to log in as the `customer` user:
 - User name: **customer**
 - Password: **redhat1!**
- 8.3. Click **Admin Page** to access the admin page. The server returns a "Forbidden" message because the `customer` user does not have proper authorization to access this page.
- 8.4. In Firefox, navigate to `about:preferences` in the URL to access the Firefox preferences page.
- 8.5. Click **Privacy** and then click **clear your recent history**. Click **Clear Now** to force the BASIC authentication to prompt you for credentials again.
- 8.6. In Firefox, navigate to `http://localhost:8080/security-domain` and enter the following credentials to log in as the `owner` user:
 - User name: **owner**
 - Password: **redhat1!**
- 8.7. In the security-domain application, click **Admin** to access the secured page of the application to view a list of users. If the security is configured correctly, the page is accessible.

- 9. Undeploy the application and stop JBoss EAP.

- 9.1. Run the following command to undeploy the application:

```
[student@workstation security-domain]$ mvn wildfly:undeploy
```

- 9.2. To close the project, right-click on the security-domain project in the **Project Explorer**, and select **Close Project**.
- 9.3. Right-click on the Red Hat JBoss EAP 7.0 server in the JBDS **Servers** tab and click **Stop**.

This concludes the guided exercise.

Securing a REST API

Objective

After completing this section, students should be able to secure a REST API with authentication and authorization.

Security with RESTEasy

One of the primary benefits of exposing an application's REST API is that the API and subsequent data becomes available to multiple application components. Exposing the API and sensitive application data, however, creates the inherent added risk of data being manipulated by unwanted users. To ensure the integrity of the application data, the REST API must be secured for both authentication and authorization. This means the API must allow only authorized users access, and it must validate the privileges of each user before manipulating data.

Similar to defining security in an EJB, a REST API is secured either with annotations or programmatically. While programmatic security provides more flexibility and control over the constraints for authorization, using annotations to define security constraints is both more readable and easier to implement. However, sometimes both programmatic implementations and security annotations are required to fully meet an organization's security requirements.

To enable role-based security for REST APIs, update the `web.xml` to contain the following `<context-param>`:

```
<context-param>
  <param-name>resteasy.role.based.security</param-name>
  <param-value>true</param-value>
</context-param>
```

After setting the `resteasy.role.based.security`, update the `web.xml` file with a security constraint set to the path of the RESTEasy service. For example, the following security constraint restricts the `/todo/api/` path to only users with the role `admin`:

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>RESTEasy</web-resource-name>
    <url-pattern>/todo/api/*</url-pattern> ①
  </web-resource-collection>
  <auth-constraint>
    <role-name>admin</role-name> ②
  </auth-constraint>
</security-constraint>
```

- ① The path to the REST API. The `*` is a wildcard. In this instance, any URL that starts with `/todo/api/` matches the security constraint.
- ② The tag to determine which role is able to access the resource.

Each role listed in the security constraint also needs to be defined in the `web.xml` as a `<security-role>`. For example, to reference the `admin` role, define the role using the following tags:

```
<security-role>
  <role-name>admin</role-name>
</security-role>
```

Complete the `web.xml` file by defining the `login-config`. As discussed in the previous section, this tag determines how the server authenticates the user and optionally defines the security realm for the application:

```
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>ApplicationRealm</realm-name>
</login-config>
```

RESTEasy Annotations

Because it can be difficult to map complex URL patterns for RESTEasy services in the `web.xml`, RESTEasy provides several annotations to secure endpoints. After enabling role-based security in the `web.xml` of the REST application, use the following annotations to secure specific endpoints by role, based on the roles listed in the `web.xml`:

- `@RolesAllowed`: Defines the role or roles that can access the method.
- `@PermitAll`: All roles defined in the `web.xml` can access the method.
- `@DenyAll`: Denies access to all roles to the method.

The following is an example of a RESTEasy class with security annotations:

```
@Stateless
@Path("hello")
@Produces(MediaType.APPLICATION_JSON)
public class HelloWorld {

    @PermitAll
    @GET
    @Produces("text/html")
    public String hello() {
        return "<b>Hello World!</b>";
    }

    @RolesAllowed("admin, guest")
    @GET
    @Path("newest")
    public Person getNewestPerson() {
        ...implementation omitted...
    }

    @RolesAllowed("admin")
    @POST
    @Consumes(MediaType.APPLICATION_JSON)
```

```

public String savePerson(Person person) {
    ...implementation omitted...
}

@DenyAll
@DELETE
public String deleteAll() {
    ...implementation omitted...
}
}

```

Demonstration: Securing a REST API

- Run the following command to prepare files used by this demonstration.

```
[student@workstation ~]$ demo secure-rest setup
```

- Start JBDS and import the `secure-rest` project.

This project is a simple RESTEasy web service that stores and reads `Person` objects from the database.

- Add the following values in the `<context-param>` to the `web.xml` to enable RESTEasy security:

```

<!-- TODO add RESTEasy context param -->
<context-param>
    <param-name>resteasy.role.based.security</param-name>
    <param-value>true</param-value>
</context-param>

```

- In the `security-constraint` applied to the entire application's URL pattern, add the roles `guest` and `admin` in the `auth-constraint` tag:

```

<auth-constraint>
    <!-- TODO add roles -->
    <role-name>guest</role-name>
    <role-name>admin</role-name>
</auth-constraint>

```

- Update the `login-config` to define the method for authentication as `BASIC`, and to use the `ApplicationRealm` as the security realm:

```

<login-config>
    <!-- TODO add login-config -->
    <auth-method>BASIC</auth-method>
    <realm-name>ApplicationRealm</realm-name>
</login-config>

```

**Note**

The ApplicationRealm is a default security realm that uses the /opt/eap/standalone/configuration/application-roles.properties properties file for roles and /opt/eap/standalone/configuration/application-users.properties properties file for users. These files are managed using the /opt/eap/bin/add-user.sh script.

Save your changes to the web.xml file using Ctrl+S.

6. Open the PersonService.java RESTEasy service class. Update each @GET method with a @PermitAll annotation to allow admin and guest users to access these methods:

```
...
@GET
// TODO add a PermitAll annotation
@PermitAll
@Path("{id}")
public Person getPerson(@PathParam("id") Long id) {
...
@GET
// TODO add a PermitAll annotation
@PermitAll
public List<Person> getAllPersons() {
...
...
```

7. Update the @Delete and @POST annotated methods to only allow the admin user authorization to run these methods:

```
...
// TODO restrict access for admin
@RolesAllowed("admin")
@DELETE
@Path("{id}")
public void deletePerson(@PathParam("id") Long id) {
...
// TODO restrict access for admin
@RolesAllowed("admin")
@POST
public Response savePerson(Person person) {
...
...
```

8. In a new terminal window, run the /opt/eap/bin/add-user.sh script to create the customer and owner users:

```
[student@workstation ~]$ cd /opt/eap/bin
[student@workstation bin]$ ./add-user.sh
```

9. Use the following information to create the customer user with the role guest:

- Application User: b

- Username: **customer**
 - Password: **redhat1!**
 - Groups: **guest**
 - About to add user 'customer' for realm 'ApplicationRealm': **yes**
 - Is this new user going to be used for one AS process to connect to another AS process?: **no**
10. Run the script again and create the **owner** user with the role **admin**:
- Application User: **b**
 - Username: **owner**
 - Password: **redhat1!**
 - Groups: **admin**
 - About to add user 'customer' for realm 'ApplicationRealm': **yes**
 - Is this new user going to be used for one AS process to connect to another AS process?: **no**
11. Start the local JBoss EAP server inside JBDS.
12. Run the following command in a terminal window to deploy the application to the local JBoss EAP server:
- ```
[student@workstation ~]$ cd /home/student/JB183/labs/secure-rest
[student@workstation secure-rest]$ mvn wildfly:deploy
```
13. Access the REST-Client plug-in in Firefox on the workstation VM. Enter `http://localhost:8080/secure-rest/api/persons` in the URL. Click **Authentication** and then click **Basic Authentication**. Log in with the following credentials:
- Username: **customer**
  - Password: **redhat1!**
- Click **Okay**.
14. Click **Send** and notice that the return code is **200**. The server returns an empty list of users in the **Response Body** tab.
15. Change the **Method Type** to **POST**. Click **Headers** in the navigation bar and then click **Custom Header**. Use the following information for the custom header and click **Okay**:
- Name: **Content-Type**
  - Value: **application/json**
16. Add the following JSON to the **Body** of the request to create a new **Person** with the name **Shadowman**:

```
{"name": "Shadowman"}
```

Click **Send** and notice that the server responds with the HTTP code 403 **Forbidden**.

17. Click **Authentication** and then click **Basic Authentication**. Log in with the following credentials:

- **Username: owner**
- **Password: redhat1!**

Click **Okay** and then click **Send** again. This time, the server responds with the HTTP code 200 **OK**.

18. Change the **Method Type** to **GET** and click **Send** again to see the new user in the **Response Body** tab:

```
[{"id":1,"name":"Shadowman"}]
```

19. Undeploy the application and stop the server.

```
[student@workstation secure-rest]$ mvn wildfly:undeploy
```



### References

Further information is available in the *Developing Web Services Applications* guide

for Red Hat JBoss EAP 7 at

<https://access.redhat.com/documentation/en/red-hat-jboss-enterprise-application-platform/>

## ► Guided Exercise

# Securing a REST API

In this exercise, you will secure a REST service for authentication and authorization.

## Outcome

You should be able to use RESTEasy security annotations to secure authentication and authorization in a REST web service.

## Before You Begin

Open a terminal window on the **workstation** VM and run the following command to download the files required for this lab.

```
[student@workstation ~]$ lab rest-annotations setup
```

## Steps

- ▶ 1. Open JBDS and import the Maven project.
  - 1.1. Open JBDS by double-clicking the JBoss Developer Studio icon on the workstation desktop. Set the workspace to `/home/student/JB183/workspace` and click **OK**.
  - 1.2. In the JBDS menu, click **File > Import** to open the **Import** wizard.
  - 1.3. On the **Select** page, click **Maven > Existing Maven Projects**, and then click **Next**.
  - 1.4. In the **Maven projects** page, click **Browse** to open the **Select root folder** window. Navigate to the `/home/student/JB183/labs/` directory. Select the `rest-annotations` folder, and then click **OK**.
  - 1.5. On the **Maven projects** page, click **Finish**.
  - 1.6. Watch the JBDS status bar to monitor the progress of the import operation. It may take a few minutes to download all required dependencies.
- ▶ 2. Start EAP by selecting the **Servers** tab in the bottom pane of JBDS. Right-click the server **Red Hat JBoss EAP 7.0 [Stopped]**, and click the green button to start the server. Watch the **Console** until the server starts and displays the message:

```
INFO [org.jboss.as] (Controller Boot Thread) WFLYSRV0025: JBoss EAP 7.0.0.GA
(WildFly Core 2.1.2.Final-redhat-1) started
```

- ▶ 3. Update the `web.xml` file to enable RESTEasy security annotations, add a security constraint for the REST service path, and use a **BASIC** authentication method.
  - 3.1. Open the `web.xml` file by expanding the `rest-annotations` item in the **Project Explorer** tab in the left pane of JBDS, then click on `rest-annotations > src/main/webapp > WEB-INF` and expand it. Double-click the `web.xml` file.

- 3.2. Create a new <context-param> tag that contains the `resteasy.role.based.security` parameter with a value of true:

```
<!-- Add context param -->
<context-param>
 <param-name>resteasy.role.based.security</param-name>
 <param-value>true</param-value>
</context-param>
```

- 3.3. Create a new `security-constraint` that restricts all resources for the application with a <url-pattern> set to `*`:

```
<!-- Add security constraint -->
<security-constraint>
 <web-resource-collection>
 <web-resource-name>All resources</web-resource-name>
 <url-pattern>/*</url-pattern>
 </web-resource-collection>
</security-constraint>
```

- 3.4. Update the `security-constraint` to restrict access for users with the guest or admin role using the `auth-constraint` tag:

```
<security-constraint>
 <web-resource-collection>
 <web-resource-name>All resources</web-resource-name>
 <url-pattern>/*</url-pattern>
 </web-resource-collection>
 <auth-constraint>
 <role-name>guest</role-name>
 <role-name>admin</role-name>
 </auth-constraint>
</security-constraint>
```

- 3.5. Define security roles for the guest and admin roles with a `security-role` tag:

```
<!-- Add security role -->
<security-role>
 <role-name>admin</role-name>
</security-role>
<security-role>
 <role-name>guest</role-name>
</security-role>
```

- 3.6. Create a `<login-config>` element and set the `<auth-method>` to BASIC and use the predefined security realm ApplicationRealm.

```
<!-- Add login config -->
<login-config>
 <auth-method>BASIC</auth-method>
 <realm-name>ApplicationRealm</realm-name>
</login-config>
```

3.7. Save your changes by pressing **Ctrl+S**.

- 4. This application now uses the **ApplicationRealm** to authenticate users. The **ApplicationRealm** is a default security realm that uses the `/opt/eap/standalone/configuration/application-roles.properties` properties file for roles and `/opt/eap/standalone/configuration/application-users.properties` properties file for users.

Use the `/opt/eap/bin/add-user.sh` script to create the following users and their respective roles:

- **customer:guest**
- **owner:admin**
- **superuser:guest,admin**

4.1. In a new terminal window, run the `/opt/eap/bin/add-user.sh` script:

```
[student@workstation ~]$ cd /opt/eap/bin
[student@workstation bin]$./add-user.sh
```

4.2. Use the following information to create the **customer** user with the role **guest** when prompted:

- Application User: **b**
- Username: **customer**
- Password: **redhat1!**
- Groups: **guest**
- About to add user 'customer' for realm 'ApplicationRealm': **yes**
- Is this new user going to be used for one AS process to connect to another AS process?: **no**

4.3. Run the script again and use the following information to create the **owner** user with the role **admin** when prompted:

- Application User: **b**
- Username: **owner**
- Password: **redhat1!**
- Groups: **admin**
- About to add user 'customer' for realm 'ApplicationRealm': **yes**
- Is this new user going to be used for one AS process to connect to another AS process?: **no**

4.4. Run the script a third time and use the following information to create the **superuser** user with both the **admin** and **guest** when prompted:

- Application User: **b**

- Username: **superuser**
  - Password: **redhat1!**
  - Groups: **admin, guest**
  - About to add user 'customer' for realm 'ApplicationRealm': **yes**
  - Is this new user going to be used for one AS process to connect to another AS process?: **no**
- 4.5. Use the following command to view the contents of the /opt/eap/standalone/configuration/application-users.properties with the following command:

```
[student@workstation ~]$ less /opt/eap/standalone/configuration\
/application-users.properties
```

```
...
customer=a2d94c2cc5f3a4706e669da24b9a9bf0
owner=2f408f4ad343d859807d7593128b3767
superuser=8e90480af06a793a083c16991b11dc99
...
```

Notice that each line contains the user and the user's hashed password.

- 4.6. Use the following command to view the contents of the /opt/eap/standalone/configuration/application-roles.properties with the following command:

```
[student@workstation ~]$ less /opt/eap/standalone/configuration\
/application-roles.properties
```

```
...
customer=guest
owner=admin
superuser=admin, guest
...
```

Notice that each line contains a user and the user's assigned role or roles.

- ▶ 5. Update the PersonService.java RESTEasy service class and update each method with a RESTEasy security annotation.
- 5.1. In the Project Explorer tab in the left pane of JBDS, select **src/main/java > com.redhat.training.rest** and double-click **PersonService.java** to view the file.
  - 5.2. Update the **getPerson()** method with a **@PermitAll** annotation to allow all roles listed in the **web.xml** to access the method:

```
//TODO permit all users
@PermitAll
@GET
@Path("{id}")
public Person getPerson(@PathParam("id") Long id) {
 ...
}
```

- 5.3. Update the `getAllPersons()` method with a `@RolesAllowed` annotation to allow only users with the role `guest` to access the method:

```
//TODO allow only guest
@RolesAllowed("guest")
@GET
public List<Person> getAllPersons() {
 ...
}
```

- 5.4. Update the `deletePerson()` `DELETE` method with a `@DenyAll` annotation to prevent all roles listed in `web.xml` from accessing this method:

```
//TODO restrict access for all roles
@DenyAll
@DELETE
@Path("{id}")
public void deletePerson(@PathParam("id") Long id) {
 ...
}
```

- 5.5. Update the `savePerson()` `POST` method with a `@RolesAllowed` annotation to allow only users with the role `admin` to access the method:

```
//TODO allow only admin
@RolesAllowed("admin")
@POST
public Response savePerson(Person person) {
 ...
}
```

- 5.6. Save your changes by pressing `Ctrl+S`.

- 6. Navigate to the project directory and deploy the application.

```
[student@workstation ~]$ cd /home/student/JB183/labs/rest-annotations
[student@workstation rest-annotations]$ mvn wildfly:deploy
```

- 7. Use the Firefox REST Client plug-in to test the secured REST API.

- 7.1. Access the REST-Client plug-in in Firefox on the workstation VM.



Figure 9.2: The Firefox REST client plug-in

- 7.2. Enter `http://localhost:8080/rest-annotations/api/persons` in the URL.
- 7.3. Click **Authentication** and then click **Basic Authentication**. Log in with the following credentials:
  - Username: **customer**
  - Password: **redhat1!**Click Okay.
- 7.4. Click **Send** and notice that the return code is **200**. Click the **Response Body** tab and notice that the payload is any empty list because no users have been added to the database yet.
- 7.5. Change the **Method Type** to **POST**.
- 7.6. Click **Headers** in the navigation bar and then click **Custom Header**. Use the following information for the custom header:
  - Name: **Content-Type**
  - Value: **application/json**Click Okay.
- 7.7. Add the following JSON to the **Body** of the request to create a new Person with the name **RedHat**:

```
{"name": "RedHat"}
```

- 7.8. Click **Send** and notice that the server responds with the HTTP code **403 Forbidden** because the user **customer** with the role **guest** is not authorized to use that method in the REST service class.
- **8.** Test the REST API again with the **owner** and **superuser** users.
- 8.1. Click **Authentication** and then click **Basic Authentication**. Log in with the following credentials:
    - Username: **owner**
    - Password: **redhat1!**Click Okay.
  - 8.2. Click **Send** using the same JSON request data from the previous step. This time, the server responds with the HTTP code **200 OK**.
  - 8.3. Verify that the data was persisted into the database by changing the **Method Type** to **GET** and then clicking **Send** again.

The server responds with another **403 Forbidden** HTTP code because the user **owner** with the role **admin** is not authorized to view all users.
  - 8.4. Click **Authentication** and then click **Basic Authentication**. Log in with the following credentials:
    - Username: **superuser**

- Password: **redhat1!**

Click **Okay**.

- 8.5. The **superuser** user belongs to both **guest** and **admin** roles and can therefore both save new **Person** objects and get a list of all objects in the database.

With the **Method Type** still set to **GET**, click **Send** again. This time, the server responds with the HTTP code **200 OK** and with the following data in the **Response Body** tab:

```
[{"id":1,"name":"RedHat"}]
```

- 8.6. While still authenticated as **superuser**, change the **Method Type** to **POST** and add the following JSON to the **Body** of the request:

```
{"name": "Shadowman"}
```

Click **Send**.

- 8.7. Verify that the data was persisted into the database by changing the **Method Type** to **GET** and then clicking **Send** again.

The server responds with the HTTP code **200 OK** and with the following data in the **Response Body** tab:

```
[{"id":1,"name":"RedHat"}, {"id":2, "name":"Shadowman"}]
```

- 8.8. Delete the **Person** in the database with the ID **1** by updating the request URL to **http://localhost:8080/rest-annotations/api/persons/1** and changing the **Method Type** to **DELETE**.

- 8.9. Click **Send**.

The server returns another **403 Forbidden** code because this method is annotated with a **@DenyAll** annotation preventing any users from executing it.

## ► 9. Undeploy the application and stop JBoss EAP.

- 9.1. Run the following command to undeploy the application:

```
[student@workstation rest-annotations]$ mvn wildfly:undeploy
```

- 9.2. To close the project, right-click on the **rest-annotations** project in the **Project Explorer** and select **Close Project**.

- 9.3. Right-click on the Red Hat JBoss EAP 7.0 server in the **JBDS Servers** tab and click **Stop**.

This concludes the guided exercise.

## ► Lab

# Securing Java EE Applications

In this lab, you will secure the To Do List application REST service with authentication and authorization.

## Outcomes

You should be able to secure a REST service with authentication and authorization.

## Before You Begin

Open a terminal window on the `workstation` VM and run the following command to download the files required for this lab.

```
[student@workstation ~]$ lab securing-lab setup
```

## Steps

1. Open JBDS and import the `securing-lab` project located in the `/home/student/JB183/labs/securing-lab` directory.
2. Start JBoss EAP from within JBDS.
3. Use the `/home/student/JB183/labs/securing-lab/create-sd.sh` script to create a `UsersRoles` security domain named `userroles`.

This security domain uses the `UsersRoles` login module to read the provided `/home/student/JB183/labs/securing-lab/todo-users.properties` user property file for authentication and the `/home/student/JB183/labs/securing-lab/todo-roles.properties` file for authorization.

- 3.1. In the terminal window, navigate to the `/home/student/JB183/labs/securing-lab` project directory and run the `create-sd.sh` script:

```
[student@workstation ~]$ cd JB183/labs/securing-lab
[student@workstation securing-lab]$./create-sd.sh
```

- 3.2. Confirm that the security domain is available by viewing the contents of the EAP server configuration `/opt/jboss-eap-7.0/standalone/configuration/standalone-full.xml`.

Using a text editor, open the `/opt/jboss-eap-7.0/standalone/configuration/standalone-full.xml` configuration file and observe the new security domain that directs the application server to use the provided users and roles property files in the project directory.

```
<security-domain name="userroles" cache-type="default">
 <authentication>
 <login-module code="UsersRoles" flag="required">
 <module-option name="usersProperties" value="file:///home/student/JB183/labs/
securing-lab/todo-users.properties"/>
 <module-option name="rolesProperties" value="file:///home/student/JB183/labs/
securing-lab/todo-users.properties"/>
 </login-module>
 </authentication>
</security-domain>
```

**4.** Update the `jboss-web.xml` file to use the `userroles` security domain.

- 4.1. Open the `jboss-web.xml` class by expanding the `securing-lab` item in the **Project Explorer** tab in the left pane of JBDS. Click on `securing-lab > src/main/webapp > WEB-INF` and expand it. Double-click the `jboss-web.xml` file.
- 4.2. Update the `jboss-web.xml` file to use the new security domain named `userroles`:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<jboss-web>
 <security-domain>userroles</security-domain>
</jboss-web>
```

- 4.3. Save your changes by pressing **Ctrl+S**.

**5.** Update the `web.xml` file to enable RESTEasy security annotations, use **BASIC** authentication, and restrict access to the application's REST API with the path `/api/*` to the roles `admin`, `guest`, and `observer`.

- 5.1. Open the `web.xml` class by expanding the `securing-lab` item in the **Project Explorer** tab in the left pane of JBDS. Click on `securing-lab > src/main/webapp > WEB-INF` and expand it. Double-click the `web.xml` file.
- 5.2. Create a new `<context-param>` tag containing the `resteasy.role.based.security` parameter with a value of `true`:

```
<!-- Add context param -->
<context-param>
 <param-name>resteasy.role.based.security</param-name>
 <param-value>true</param-value>
</context-param>
```

- 5.3. Create a new `<security-constraint>` that restricts all resources for the application with an `<url-pattern>` set to `/api/*`:

```
<!-- Add security constraint -->
<security-constraint>
 <web-resource-collection>
 <web-resource-name>REST Resources</web-resource-name>
 <url-pattern>/api/*</url-pattern>
 </web-resource-collection>
</security-constraint>
```

- 5.4. Update the `<security-constraint>` to restrict access for users with the `guest`, `observer`, or `admin` role using the `auth-constraint` tag:

```
<security-constraint>
 <web-resource-collection>
 <web-resource-name>All resources</web-resource-name>
 <url-pattern>/api/*</url-pattern>
 </web-resource-collection>
 <auth-constraint>
 <role-name>guest</role-name>
 <role-name>admin</role-name>
 <role-name>observer</role-name>
 </auth-constraint>
</security-constraint>
```

- 5.5. Define the security roles for the `guest`, `observer`, and `admin` roles with a `<security-role>` tag:

```
<!-- Add security role -->
<security-role>
 <role-name>admin</role-name>
</security-role>
<security-role>
 <role-name>guest</role-name>
</security-role>
<security-role>
 <role-name>observer</role-name>
</security-role>
```

- 5.6. Create a `<login-config>` element and set the `<auth-method>` to `BASIC`.

```
<!-- Add login config -->
<login-config>
 <auth-method>BASIC</auth-method>
</login-config>
```

- 5.7. Save your changes by pressing `Ctrl+S`.

6. Update the `ItemResourceRESTService.java` RESTEasy service class and update each method with the following requirements:
- All GET methods are permitted for all roles.
  - All POST methods are permitted for `admin` and `guest` roles.

- All DELETE methods are permitted only for users with the role admin.

Notice that the `@PostConstruct` method retrieves the `Principal` user from the request in order to determine which user is currently logged in. This way, the REST service only retrieves results for that specific user.

- 6.1. In the Project Explorer tab in the left pane of JBDS, select `src/main/java > com.redhat.training.rest`. Double-click `ItemResourceRESTService.java` to view the file.
- 6.2. Update the `listAllItems` method with a `@PermitAll` annotation to allow all roles listed in the `web.xml` to access the method:

```
@PermitAll
@GET
@Produces(MediaType.APPLICATION_JSON)
public List<Item> listAllItems() {
 return repository.findAllItemsForUser(currentUser);
}
```

- 6.3. Update the `lookupItemById()` method with a `@PermitAll` annotation to allow all roles listed in `web.xml` to access the method:

```
@PermitAll
@GET
@Path("/{id:[0-9][0-9]*}")
@Produces(MediaType.APPLICATION_JSON)
public Item lookupItemById(@PathParam("id") long id) {
 ...
}
```

- 6.4. Update the `createItem()` POST method with a `@RolesAllowed` annotation to allow only users with the role admin or guest to access the method:

```
@RolesAllowed({"admin", "guest"})
@POST
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response createItem(Item item) {
 ...
}
```

- 6.5. Update the `deleteItem()` DELETE method with a `@RolesAllowed` annotation to allow only users with the role admin to access the method:

```
@RolesAllowed("admin")
@DELETE
@Path("{id}")
@Consumes(MediaType.APPLICATION_JSON)
public void deleteItem(@PathParam("id") Long id) {
 itemService.remove(id);
}
```

- 6.6. Save your changes by pressing `Ctrl+S`.

7. Build and deploy the application to JBoss EAP using Maven by running the following commands:
8. Use the Firefox REST Client plug-in to test the secured REST API at the URL `http://localhost:8080/securing-lab/api/items`. Use the plug-in to verify that the `customer` user with password `redhat1!` has permissions to access the GET and POST methods only.

8.1. Access the REST-Client plug-in in Firefox on the workstation VM.



**Figure 9.3: The Firefox REST client plug-in**

- 8.2. Enter `http://localhost:8080/securing-lab/api/items` in the URL.
- 8.3. Click **Authentication** and then click **Basic Authentication**. Log in with the following credentials:
  - Username: **customer**
  - Password: **redhat1!**
 Click **Okay**.
- 8.4. Change the **Method Type** to **POST**.
- 8.5. Click **Headers** in the navigation bar and then click **Custom Header**. Use the following information for the custom header:
  - Name: **Content-Type**
  - Value: **application/json**
 Click **Okay**.
- 8.6. Add the following JSON to the **Body** of the request to create a new **Item** with the description `Walk the dog`:

```
{"description": "Walk the dog"}
```

Click **Send**.

- 8.7. Change the **Method Type** to **GET** and click **Send** again to see a list of all of the **Item** objects:

```
[{"id":11, "description":"Walk the dog", "done":false,"user":{"id":2,"username":"customer"}}]
```

- 8.8. Using the ID from the previous response, enter `http://localhost:8080/securing-lab/api/items/11` in the URL of the plug-in and change the **Method Type** to **DELETE**. Click **Send**, and the server returns a **403 Forbidden** response.
9. Use the plug-in to verify that the `owner` user with password `redhat1!` has permissions to access the **GET**, **POST**, and **DELETE** methods.

- 9.1. In the Firefox plug-in, click **Authentication** and then click **Basic Authentication**. Log in with the following credentials:
  - Username: **owner**
  - Password: **redhat1!**Click **Okay**.
- 9.2. Change the **Method Type** to **POST** and update the URL to be `http://localhost:8080/securing-lab/api/items/`.
- 9.3. Add the following JSON to the **Body** of the request to create a new **Item** with the description `Test owner`:

```
{"description": "Test owner"}
```
- 9.4. Change the **Method Type** to **GET** and click **Send** again to see a list of all of the **Item** objects:

```
[{"id":12, "description":"Test owner", "done":false, "user":{"id":3, "username":"owner"}}]
```
- 9.5. Using the ID from the previous response, enter `http://localhost:8080/securing-lab/api/items/12` in the URL of the plug-in and change the **Method Type** to **DELETE**. Click **Send**, and the server returns a **204** response.
- 9.6. Change the **Method Type** to **GET**, update the URL to `http://localhost:8080/securing-lab/api/items/` and click **Send** again to see that the **Item** object is removed.
10. Use the plug-in to verify that the **viewer** user with password **redhat1!** has permissions to access only the **GET** methods.
  - 10.1. In the Firefox plug-in, click **Authentication** and then click **Basic Authentication**. Log in with the following credentials:
    - Username: **viewer**
    - Password: **redhat1!**Click **Okay**.
  - 10.2. Change the **Method Type** to **POST**.
  - 10.3. Add the following JSON to the **Body** of the request to create a new **Item** with the description `Test viewer`:

```
{"description": "Test viewer"}
```
  - 10.4. Change the **Method Type** to **GET** and click **Send** again to see a **200 Ok** response from the server, although the list is empty.

11. Open a new terminal window and run the following command to grade the lab:

```
[student@workstation ~]$ lab securing-lab grade
```

The grading script should report SUCCESS. If there is a failure, check the errors and fix them until you see a SUCCESS message.

12. Clean up.

12.1. Undeploy the application from JBoss EAP using Maven with the following commands:

```
[student@workstation ~]$ cd /home/student/JB183/labs/securing-lab
[student@workstation securing-lab]$ mvn wildfly:undeploy
```

12.2. Right-click on the securing-lab project in the **Project Explorer**, and select **Close Project** to close this project.

12.3. Right-click on the Red Hat JBoss EAP 7.0 server in the **JBDS Servers** tab and click **Stop**.

This concludes the lab.

## ► Solution

# Securing Java EE Applications

In this lab, you will secure the To Do List application REST service with authentication and authorization.

### Outcomes

You should be able to secure a REST service with authentication and authorization.

### Before You Begin

Open a terminal window on the **workstation** VM and run the following command to download the files required for this lab.

```
[student@workstation ~]$ lab securing-lab setup
```

### Steps

1. Open JBDS and import the **securing-lab** project located in the `/home/student/JB183/labs/securing-lab` directory.
  - 1.1. To open JBDS, double-click the JBoss Developer Studio icon on the workstation desktop. Leave the default workspace (`/home/student/JB183/workspace`) and click **OK**.
  - 1.2. In the JBDS menu, click **File > Import** to open the **Import** wizard.
  - 1.3. On the **Select** page, click **Maven > Existing Maven Projects**, and then click **Next**.
  - 1.4. On the **Maven projects** page, click **Browse** to open the **Select root folder** window. Navigate to the `/home/student/JB183/labs/` directory. Select the **securing-lab** folder, and then click **OK**.
  - 1.5. On the **Maven projects** page, click **Finish**.
  - 1.6. Watch the JBDS status bar to monitor the progress of the import operation. It may take a few minutes to download all the required dependencies.
2. Start JBoss EAP from within JBDS.  
Select the **Servers** tab in JBDS. Right-click on the server entry **Red Hat JBoss EAP 7.0 [Stopped]** and click the green **Start** option to start the server.
3. Use the `/home/student/JB183/labs/securing-lab/create-sd.sh` script to create a **UsersRoles** security domain named **userroles**.  
This security domain uses the **UsersRoles** login module to read the provided `/home/student/JB183/labs/securing-lab/todo-users.properties` user property file for authentication and the `/home/student/JB183/labs/securing-lab/todo-roles.properties` file for authorization.
  - 3.1. In the terminal window, navigate to the `/home/student/JB183/labs/securing-lab/` project directory and run the `create-sd.sh` script:

```
[student@workstation ~]$ cd JB183/labs/securing-lab
[student@workstation securing-lab]$./create-sd.sh
```

- 3.2. Confirm that the security domain is available by viewing the contents of the EAP server configuration /opt/jboss-eap-7.0/standalone/configuration/standalone-full.xml.

Using a text editor, open the /opt/jboss-eap-7.0/standalone/configuration/standalone-full.xml configuration file and observe the new security domain that directs the application server to use the provided users and roles property files in the project directory.

```
<security-domain name="userroles" cache-type="default">
 <authentication>
 <login-module code="UsersRoles" flag="required">
 <module-option name="usersProperties" value="file:///home/student/JB183/labs/securing-lab/todo-users.properties"/>
 <module-option name="rolesProperties" value="file:///home/student/JB183/labs/securing-lab/todo-users.properties"/>
 </login-module>
 </authentication>
</security-domain>
```

4. Update the jboss-web.xml file to use the userroles security domain.

- 4.1. Open the jboss-web.xml class by expanding the securing-lab item in the Project Explorer tab in the left pane of JBDS. Click on securing-lab > src/main/webapp > WEB-INF and expand it. Double-click the jboss-web.xml file.

- 4.2. Update the jboss-web.xml file to use the new security domain named userroles:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<jboss-web>
 <security-domain>userroles</security-domain>
</jboss-web>
```

- 4.3. Save your changes by pressing Ctrl+S.

5. Update the web.xml file to enable RESTEasy security annotations, use BASIC authentication, and restrict access to the application's REST API with the path /api/\* to the roles admin, guest, and observer.

- 5.1. Open the web.xml class by expanding the securing-lab item in the Project Explorer tab in the left pane of JBDS. Click on securing-lab > src/main/webapp > WEB-INF and expand it. Double-click the web.xml file.

- 5.2. Create a new <context-param> tag containing the resteasy.role.based.security parameter with a value of true:

```
<!-- Add context param -->
<context-param>
 <param-name>resteasy.role.based.security</param-name>
 <param-value>true</param-value>
</context-param>
```

- 5.3. Create a new **security-constraint** that restricts all resources for the application with an **<url-pattern>** set to **/api/\***:

```
<!-- Add security constraint -->
<security-constraint>
 <web-resource-collection>
 <web-resource-name>REST Resources</web-resource-name>
 <url-pattern>/api/*</url-pattern>
 </web-resource-collection>
</security-constraint>
```

- 5.4. Update the **<security-constraint>** to restrict access for users with the **guest**, **observer**, or **admin** role using the **auth-constraint** tag:

```
<security-constraint>
 <web-resource-collection>
 <web-resource-name>All resources</web-resource-name>
 <url-pattern>/api/*</url-pattern>
 </web-resource-collection>
 <auth-constraint>
 <role-name>guest</role-name>
 <role-name>admin</role-name>
 <role-name>observer</role-name>
 </auth-constraint>
</security-constraint>
```

- 5.5. Define the security roles for the **guest**, **observer**, and **admin** roles with a **<security-role>** tag:

```
<!-- Add security role -->
<security-role>
 <role-name>admin</role-name>
</security-role>
<security-role>
 <role-name>guest</role-name>
</security-role>
<security-role>
 <role-name>observer</role-name>
</security-role>
```

- 5.6. Create a **<login-config>** element and set the **<auth-method>** to **BASIC**.

```
<!-- Add login config -->
<login-config>
 <auth-method>BASIC</auth-method>
</login-config>
```

- 5.7. Save your changes by pressing **Ctrl+S**.
6. Update the `ItemResourceRESTService.java` RESTEasy service class and update each method with the following requirements:
- All GET methods are permitted for all roles.
  - All POST methods are permitted for `admin` and `guest` roles.
  - All DELETE methods are permitted only for users with the role `admin`.

Notice that the `@PostConstruct` method retrieves the `Principal` user from the request in order to determine which user is currently logged in. This way, the REST service only retrieves results for that specific user.

- 6.1. In the **Project Explorer** tab in the left pane of JBDS, select `src/main/java > com.redhat.training.rest`. Double-click `ItemResourceRESTService.java` to view the file.
- 6.2. Update the `listAllItems` method with a `@PermitAll` annotation to allow all roles listed in the `web.xml` to access the method:

```
@PermitAll
@GET
@Produces(MediaType.APPLICATION_JSON)
public List<Item> listAllItems() {
 return repository.findAllItemsForUser(currentUser);
}
```

- 6.3. Update the `lookupItemById()` method with a `@PermitAll` annotation to allow all roles listed in `web.xml` to access the method:

```
@PermitAll
@GET
@Path("/{id:[0-9][0-9]*}")
@Produces(MediaType.APPLICATION_JSON)
public Item lookupItemById(@PathParam("id") long id) {
 ...
}
```

- 6.4. Update the `createItem()` POST method with a `@RolesAllowed` annotation to allow only users with the role `admin` or `guest` to access the method:

```
@RolesAllowed({"admin", "guest"})
@POST
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response createItem(Item item) {
 ...
}
```

- 6.5. Update the `deleteItem()` DELETE method with a `@RolesAllowed` annotation to allow only users with the role `admin` to access the method:

```
@RolesAllowed("admin")
@DELETE
@Path("{id}")
@Consumes(MediaType.APPLICATION_JSON)
public void deleteItem(@PathParam("id") Long id) {
 itemService.remove(id);
}
```

- 6.6. Save your changes by pressing `Ctrl+S`.

7. Build and deploy the application to JBoss EAP using Maven by running the following commands:

```
[student@workstation ~]$ cd /home/student/JB183/labs/securing-lab
[student@workstation securing-lab]$ mvn clean wildfly:deploy
```

8. Use the Firefox REST Client plug-in to test the secured REST API at the URL `http://localhost:8080/securing-lab/api/items`. Use the plug-in to verify that the `customer` user with password `redhat1!` has permissions to access the GET and POST methods only.

- 8.1. Access the REST-Client plug-in in Firefox on the workstation VM.



Figure 9.4: The Firefox REST client plug-in

- 8.2. Enter `http://localhost:8080/securing-lab/api/items` in the URL.
- 8.3. Click **Authentication** and then click **Basic Authentication**. Log in with the following credentials:
- Username: **customer**
  - Password: **redhat1!**
- Click **Okay**.
- 8.4. Change the **Method Type** to **POST**.
- 8.5. Click **Headers** in the navigation bar and then click **Custom Header**. Use the following information for the custom header:
- Name: **Content-Type**
  - Value: **application/json**
- Click **Okay**.
- 8.6. Add the following JSON to the **Body** of the request to create a new **Item** with the description **Walk the dog**:

```
{"description": "Walk the dog"}
```

Click **Send**.

- 8.7. Change the **Method Type** to **GET** and click **Send** again to see a list of all of the **Item** objects:

```
[{"id":11, "description":"Walk the dog", "done":false, "user":{"id":2, "username":"customer"}}]
```

- 8.8. Using the ID from the previous response, enter `http://localhost:8080/securing-lab/api/items/11` in the URL of the plug-in and change the **Method Type** to **DELETE**. Click **Send**, and the server returns a **403 Forbidden** response.
9. Use the plug-in to verify that the **owner** user with password **redhat1!** has permissions to access the **GET**, **POST**, and **DELETE** methods.
- 9.1. In the Firefox plug-in, click **Authentication** and then click **Basic Authentication**. Log in with the following credentials:
- **Username:** **owner**
  - **Password:** **redhat1!**
- Click **Okay**.
- 9.2. Change the **Method Type** to **POST** and update the URL to be `http://localhost:8080/securing-lab/api/items/`.
- 9.3. Add the following JSON to the **Body** of the request to create a new **Item** with the description **Test owner**:

```
{"description": "Test owner"}
```

Click **Send**.

- 9.4. Change the **Method Type** to **GET** and click **Send** again to see a list of all of the **Item** objects:

```
[{"id":12, "description":"Test owner", "done":false, "user":{"id":3, "username":"owner"}}]
```

- 9.5. Using the ID from the previous response, enter `http://localhost:8080/securing-lab/api/items/12` in the URL of the plug-in and change the **Method Type** to **DELETE**. Click **Send**, and the server returns a **204** response.
- 9.6. Change the **Method Type** to **GET**, update the URL to `http://localhost:8080/securing-lab/api/items/` and click **Send** again to see that the **Item** object is removed.
10. Use the plug-in to verify that the **viewer** user with password **redhat1!** has permissions to access only the **GET** methods.
- 10.1. In the Firefox plug-in, click **Authentication** and then click **Basic Authentication**. Log in with the following credentials:

- Username: **viewer**
- Password: **redhat1!**

Click **Okay**.

10.2. Change the **Method Type** to **POST**.

10.3. Add the following JSON to the **Body** of the request to create a new **Item** with the description **Test viewer**:

```
{"description": "Test viewer"}
```

Click **Send**. The server returns a **403 Forbidden** code.

10.4. Change the **Method Type** to **GET** and click **Send** again to see a **200 Ok** response from the server, although the list is empty.

11. Open a new terminal window and run the following command to grade the lab:

```
[student@workstation ~]$ lab securing-lab grade
```

The grading script should report **SUCCESS**. If there is a failure, check the errors and fix them until you see a **SUCCESS** message.

12. Clean up.

12.1. Undeploy the application from JBoss EAP using Maven with the following commands:

```
[student@workstation ~]$ cd /home/student/JB183/labs/securing-lab
[student@workstation securing-lab]$ mvn wildfly:undeploy
```

12.2. Right-click on the **securing-lab** project in the **Project Explorer**, and select **Close Project** to close this project.

12.3. Right-click on the Red Hat JBoss EAP 7.0 server in the JBDS **Servers** tab and click **Stop**.

This concludes the lab.

# Summary

---

In this chapter, you learned:

- Java Authentication and Authorization Service (JAAS) is a security API that is used to implement user authentication and authorization in Java applications.
- Declarative security leverages deployment descriptors and annotations to define security behavior, while programmatic security uses libraries to manage user authentication and authorization.
- The `web.xml` and `jboss-web.xml` are deployment descriptors used to define security behavior for an application.
- In EAP, a security domain defines a set of JAAS declarative security configurations.
- The other default security domain in EAP uses the `ApplicationRealm`, which stores user information in the `application-users.properties` and `application-roles.properties` property files.
- To use the RESTEasy security annotations, add the `resteasy.role.based.security` context parameter to the application's `web.xml` file.
- RESTEasy provides the following annotations for securing a REST service with declarative security:
  - `@DenyAll`
  - `@PermitAll`
  - `@RolesAllowed`

## Chapter 10

# Comprehensive Review: Red Hat Application Development I: Programming in Java EE

### Overview

<b>Goal</b>	Review tasks from <i>Red Hat Application Development I: Programming in Java EE</i>
<b>Objectives</b>	<ul style="list-style-type: none"><li>Review tasks from <i>Red Hat Application Development I: Programming in Java EE</i></li></ul>
<b>Sections</b>	<ul style="list-style-type: none"><li>Comprehensive Review</li></ul>
<b>Lab</b>	<ul style="list-style-type: none"><li>Lab: Creating an API using JAX-RS</li><li>Lab: Persisting Data with JPA</li><li>Lab: Securing the REST API with JAAS</li></ul>

# Comprehensive Review

---

## Objectives

After completing this section, students should be able to review and refresh knowledge and skills learned in *Red Hat Application Development I: Programming in Java EE*.

### Reviewing Red Hat Application Development I: Programming in Java EE

Before beginning the comprehensive review for this course, students should be comfortable with the topics covered in each chapter.

Students can refer to earlier sections in the textbook for extra study.

#### **Chapter 1, Transitioning to Multi-tiered Applications**

Describe Java EE features and distinguish between Java EE and Java SE applications.

- Describe "enterprise application" and name some of the benefits of Java EE applications.
- Compare the features of Java EE to Java SE.
- Describe the specifications and version numbers for Java EE 7 and the process used to introduce new and updated APIs into Java EE.
- Describe various multi-tiered architectures.
- Install JBoss Developer Studio, Maven, and JBoss Enterprise Application Platform.

#### **Chapter 2, Packaging and Deploying a Java EE Application**

Describe the architecture of a Java EE application server, package an application, and deploy the application to an EAP server.

- Identify the key features of application servers and describe the Java EE server architecture.
- List the most common JNDI resource types and their typical naming convention.
- Package a simple Java EE application and deploy it to JBoss EAP using Maven.

#### **Chapter 3, Creating Enterprise Java Beans**

Create Enterprise Java Beans.

- Convert a POJO to an EJB.
- Access an EJB both locally and remotely.
- Describe the life cycle of EJBs.
- Describe container-managed and bean-managed transactions and demarcate each in an EJB.

## **Chapter 4, Managing Persistence**

Create Persistence Entities with validations.

- Describe the Persistence API.
- Persist data to a data store using entities.
- Annotate beans to validate data.
- Create a query using the Java Persistence Query Language.

## **Chapter 5, Managing Entity Relationships**

Define and manage JPA entity relationships.

- Configure one-to-one and one-to-many entity relationships.
- Describe many-to-many entity relationships.

## **Chapter 6, Creating REST Services**

Create REST APIs using the JAX-RS specification.

- Describe web services concepts and list types of web services.
- Create a REST service using the JAX-RS specification.
- Create a client application that can invoke REST APIs remotely.

## **Chapter 7, Implementing Contexts and Dependency Injection**

Describe typical use cases for using CDI, and successfully implement it in an application.

- Describe resource injection, dependency injection, and the differences between them.
- Apply scopes to beans appropriately.

## **Chapter 8, Creating Messaging Applications with JMS**

Create messaging clients that send and receive messages using the JMS API.

- Describe the JMS API and name the objects used in sending and receiving messages.
- Describe the components that make up the JMS API.
- Create a JMS client that produces and consumes messages using the JMS API.
- Create, package, and deploy a message driven bean.

## **Chapter 9, Securing Java EE Applications**

Secure a Java EE application with JAAS.

- Describe the JAAS specification.
- Configure a security domain in the JBoss EAP application server.
- Secure a REST API with authentication and role-based authorization.

## ► Lab

# Creating an API using JAX-RS

In this review, you will create a REST API using JAX-RS that injects an EJB with place-holder code for creating, reading, updating and deleting (CRUD) operations.

## Outcomes

You should be able to:

- Create an EJB that is available for injection.
- Expose a REST API that consumes and produces JSON data using JAX-RS annotations.
- Inject the EJB that was created to provide business logic needed by the REST API.

## Before You Begin

If you did not reset `workstation` and `services` at the end of the last chapter, save any work you want to keep from earlier exercises from those machines and do so now.

Set up your computers for this exercise by logging into `workstation` as `student`, and running the following command:

```
[student@workstation ~]$ lab jaxrs-review setup
```

## Instructions

1. Open JBDS, and import the `jaxrs-review` project skeleton. Review the existing code and notice the following:
  - The `Employee` model object in the `com.redhat.training.model` package represents a basic employee record, with attributes for an `id` and a `name`.
  - The `EmployeeLogger` EJB class in the `com.redhat.training.util` package provides a utility method `logAction(Employee employee, Operation operation)` to log actions taken on an employee record to a special log file.
2. Create an EJB to provide stubbed methods that implement the CRUD operations for an `Employee` object. In this exercise, these methods do not use any actual persistence. The EJB must meet the following requirements:
  - The EJB's class name is `EmployeeBean`
  - The `EmployeeBean` EJB is stateless and available for injection.
  - The `EmployeeBean` EJB injects an instance of the `EmployeeLogger` EJB.
  - The `EmployeeBean` EJB class implements four methods:
    - `createEmployee(Employee e)`
    - `readEmployeeById(Long id)`
    - `updateEmployee(Employee e)`

- `deleteEmployee(Employee e)`
  - Each of the methods on the `EmployeeBean` EJB is only a stub, and does not contain actual persistence functionality yet. Instead, each method uses the injected `EmployeeLogger` utility class to print a message that simulates its operation.
  - The `createEmployee` method stub calls the `logAction` method on the `EmployeeLogger` class, and passes in the employee object being created and an operation value of `CREATE`.
  - The `readEmployeeById` method stub creates a new employee object with the name `Example Employee`, and the ID that was passed into the method. The creation of this object must happen before the `EmployeeLogger` is called. The method must call the `logAction` method and pass in this employee object and an operation value of `READ`. Finally, the method must return the `Employee` object that was created.
  - The `updateEmployee` method stub calls the `logAction` method on the `EmployeeLogger` class, and passes in the employee object being updated and an operation value of `UPDATE`.
  - The `deleteEmployee` method stub calls the `logAction` method on the `EmployeeLogger` class, and passes in the employee object being deleted and an operation value of `DELETE`.
3. Activate JAX-RS to enable REST services to be deployed as part of the `jaxrs-review` application. All REST services that are implemented should be accessed under the path `/api`. This means the entire URL to access REST services deployed in this application matches the following, where `service_path` is specific to each REST service:

```
http://localhost:8080/jaxrs-review/api/service_path
```

4. Create a class that implements a REST service that injects the EJB, and uses its CRUD methods to provide a REST API to manage the employee data. The REST service must meet the following requirements:
- The REST service class name is `EmployeeRestService`
  - The REST service is a state-less EJB.
  - The relative path to this REST service is `employees`
  - The REST API produces and consumes JSON data.
  - The `EmployeeRestService` class injects an instance of the `EmployeeBean` named `employeeBean`, which it uses to implement the business logic of these three REST API methods.
  - The REST API that the `EmployeeRestService` service provides includes three methods, each associated with a different HTTP method. These are summarized in the following table:

#### **EmployeeRestService Method Summary**

<b>Method Signature</b>	<b>HTTP method</b>
<code>public Employee getEmployee(Long id)</code>	GET

Method Signature	HTTP method
<code>public void deleteEmployee(Long id)</code>	DELETE
<code>public Response saveEmployee(Employee employee)</code>	POST

- The `getEmployee` method returns JSON data for the `Employee` object that has a specific ID value. This method directly calls the `readEmployeeById` method on the `EmployeeBean` instance, passing in the ID of the employee, and returning the result. The ID of the employee is automatically mapped using a path-parameter.

For example, the URL to get the employee with an ID value of 1:

```
http://localhost:8080/jaxrs-review/api/employees/1
```

- The `deleteEmployee` method returns no data, and uses the `deleteEmployee` method on the `EmployeeBean` to delete the employee with a given ID value. The ID of the employee is automatically mapped using a path-parameter. The method should use the `readEmployeeById` method on the `EmployeeBean` EJB to retrieve the employee record by its ID, then call the `deleteEmployee` method to delete the retrieved employee record.
- The `saveEmployee` method can either create a new employee or update an existing employee. This method consumes a JSON representation of an `Employee` object, which is automatically mapped into one of the parameters of the method. It should return a `javax.ws.rs.core.Response` object. **After you have defined the method and the necessary annotations, use the code snippet located in `saveEmployee.txt` to implement the logic of the method.**
- Notice that the `saveEmployee` method actually calls either `createEmployee` or `updateEmployee`, based on whether or not an `Employee` record has an ID value. If an ID is specified, the `readEmployeeById` method is used to verify that an employee with that ID already exists, otherwise a server error is thrown.

- Start JBoss EAP, and use Maven to deploy the `jaxrs-review` application.
- Test the REST service and all of its methods using the REST Client Firefox plugin.
  - Specify a custom header with the name `Content-Type` and the value `application/json`.
  - Use an HTTP GET to test the `getEmployee` method and specify any ID value. Check the `EmployeeLog.txt` file, or the JBoss EAP server logs to make sure the REST service and EJB are functioning as expected.
  - Use an HTTP DELETE to test the `deleteEmployee` method and specify any ID value. Check the `EmployeeLog.txt` file, or the JBoss EAP server logs to make sure the REST service and EJB are functioning as expected.
  - Use an HTTP POST, and include the contents of the `Employee.json` file to test the `saveEmployee` method. This data includes an ID, so it triggers an employee read operation and then an update. Check the `EmployeeLog.txt` file, or the JBoss EAP server logs to make sure that the REST service and EJB are functioning as expected.

- Remove the ID value from the JSON data, and then send another HTTP POST to test the `saveEmployee` method. This triggers an employee create operation. Check the `EmployeeLog.txt` file, or the JBoss EAP server logs to make sure the REST service and EJB are functioning as expected.

## Steps

### Evaluation

As the student user on `workstation`, run the `lab_jaxrs-review` script with the `grade` argument, to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab_jaxrs-review grade
```

After the grading succeeds, undeploy the project, stop the EAP server and close the project in JBDS.

This concludes the lab.

## ► Solution

# Creating an API using JAX-RS

In this review, you will create a REST API using JAX-RS that injects an EJB with place-holder code for creating, reading, updating and deleting (CRUD) operations.

## Outcomes

You should be able to:

- Create an EJB that is available for injection.
- Expose a REST API that consumes and produces JSON data using JAX-RS annotations.
- Inject the EJB that was created to provide business logic needed by the REST API.

## Before You Begin

If you did not reset `workstation` and `services` at the end of the last chapter, save any work you want to keep from earlier exercises from those machines and do so now.

Set up your computers for this exercise by logging into `workstation` as `student`, and running the following command:

```
[student@workstation ~]$ lab jaxrs-review setup
```

## Instructions

1. Open JBDS, and import the `jaxrs-review` project skeleton. Review the existing code and notice the following:
  - The `Employee` model object in the `com.redhat.training.model` package represents a basic employee record, with attributes for an `id` and a `name`.
  - The `EmployeeLogger` EJB class in the `com.redhat.training.util` package provides a utility method `logAction(Employee employee, Operation operation)` to log actions taken on an employee record to a special log file.
2. Create an EJB to provide stubbed methods that implement the CRUD operations for an `Employee` object. In this exercise, these methods do not use any actual persistence. The EJB must meet the following requirements:
  - The EJB's class name is `EmployeeBean`
  - The `EmployeeBean` EJB is stateless and available for injection.
  - The `EmployeeBean` EJB injects an instance of the `EmployeeLogger` EJB.
  - The `EmployeeBean` EJB class implements four methods:
    - `createEmployee(Employee e)`
    - `readEmployeeById(Long id)`
    - `updateEmployee(Employee e)`

- `deleteEmployee(Employee e)`
  - Each of the methods on the `EmployeeBean` EJB is only a stub, and does not contain actual persistence functionality yet. Instead, each method uses the injected `EmployeeLogger` utility class to print a message that simulates its operation.
  - The `createEmployee` method stub calls the `logAction` method on the `EmployeeLogger` class, and passes in the employee object being created and an operation value of `CREATE`.
  - The `readEmployeeById` method stub creates a new employee object with the name `Example Employee`, and the ID that was passed into the method. The creation of this object must happen before the `EmployeeLogger` is called. The method must call the `logAction` method and pass in this employee object and an operation value of `READ`. Finally, the method must return the `Employee` object that was created.
  - The `updateEmployee` method stub calls the `logAction` method on the `EmployeeLogger` class, and passes in the employee object being updated and an operation value of `UPDATE`.
  - The `deleteEmployee` method stub calls the `logAction` method on the `EmployeeLogger` class, and passes in the employee object being deleted and an operation value of `DELETE`.
3. Activate JAX-RS to enable REST services to be deployed as part of the `jaxrs-review` application. All REST services that are implemented should be accessed under the path `/api`. This means the entire URL to access REST services deployed in this application matches the following, where `service_path` is specific to each REST service:

```
http://localhost:8080/jaxrs-review/api/service_path
```

4. Create a class that implements a REST service that injects the EJB, and uses its CRUD methods to provide a REST API to manage the employee data. The REST service must meet the following requirements:
- The REST service class name is `EmployeeRestService`
  - The REST service is a state-less EJB.
  - The relative path to this REST service is `employees`
  - The REST API produces and consumes JSON data.
  - The `EmployeeRestService` class injects an instance of the `EmployeeBean` named `employeeBean`, which it uses to implement the business logic of these three REST API methods.
  - The REST API that the `EmployeeRestService` service provides includes three methods, each associated with a different HTTP method. These are summarized in the following table:

#### EmployeeRestService Method Summary

Method Signature	HTTP method
<code>public Employee getEmployee(Long id)</code>	GET

Method Signature	HTTP method
<code>public void deleteEmployee(Long id)</code>	DELETE
<code>public Response saveEmployee(Employee employee)</code>	POST

- The `getEmployee` method returns JSON data for the `Employee` object that has a specific ID value. This method directly calls the `readEmployeeById` method on the `EmployeeBean` instance, passing in the ID of the employee, and returning the result. The ID of the employee is automatically mapped using a path-parameter.

For example, the URL to get the employee with an ID value of 1:

```
http://localhost:8080/jaxrs-review/api/employees/1
```

- The `deleteEmployee` method returns no data, and uses the `deleteEmployee` method on the `EmployeeBean` to delete the employee with a given ID value. The ID of the employee is automatically mapped using a path-parameter. The method should use the `readEmployeeById` method on the `EmployeeBean` EJB to retrieve the employee record by its ID, then call the `deleteEmployee` method to delete the retrieved employee record.
- The `saveEmployee` method can either create a new employee or update an existing employee. This method consumes a JSON representation of an `Employee` object, which is automatically mapped into one of the parameters of the method. It should return a `javax.ws.rs.core.Response` object. **After you have defined the method and the necessary annotations, use the code snippet located in `saveEmployee.txt` to implement the logic of the method.**
- Notice that the `saveEmployee` method actually calls either `createEmployee` or `updateEmployee`, based on whether or not an `Employee` record has an ID value. If an ID is specified, the `readEmployeeById` method is used to verify that an employee with that ID already exists, otherwise a server error is thrown.

- Start JBoss EAP, and use Maven to deploy the `jaxrs-review` application.
- Test the REST service and all of its methods using the REST Client Firefox plugin.
  - Specify a custom header with the name `Content-Type` and the value `application/json`.
  - Use an HTTP GET to test the `getEmployee` method and specify any ID value. Check the `EmployeeLog.txt` file, or the JBoss EAP server logs to make sure the REST service and EJB are functioning as expected.
  - Use an HTTP DELETE to test the `deleteEmployee` method and specify any ID value. Check the `EmployeeLog.txt` file, or the JBoss EAP server logs to make sure the REST service and EJB are functioning as expected.
  - Use an HTTP POST, and include the contents of the `Employee.json` file to test the `saveEmployee` method. This data includes an ID, so it triggers an employee read operation and then an update. Check the `EmployeeLog.txt` file, or the JBoss EAP server logs to make sure that the REST service and EJB are functioning as expected.

- Remove the ID value from the JSON data, and then send another HTTP POST to test the `saveEmployee` method. This triggers an employee create operation. Check the `EmployeeLog.txt` file, or the JBoss EAP server logs to make sure the REST service and EJB are functioning as expected.

## Steps

### Steps

- Open JBDS and import the Maven project.
  - Open JBDS by double-clicking the JBoss Developer Studio icon on the workstation desktop. Set the workspace to `/home/student/JB183/workspace` and click **OK**.
  - In the JBDS menu, click **File > Import** to open the **Import** wizard.
  - On the **Select** page, click **Maven > Existing Maven Projects**, and then click **Next**.
  - In the **Maven projects** page, click **Browse** to open the **Select root folder** window. Navigate to the `/home/student/JB183/labs/` directory. Select the `jaxrs-review` folder, and then click **OK**.
  - On the **Maven projects** page, click **Finish**.
  - Watch the JBDS status bar to monitor the progress of the import operation. It may take a few minutes to download all the required dependencies.
- Create an EJB to provide the stubbed methods for the CRUD operations for an Employee object.
  - Right-click `com.redhat.training.ejb`, and click **New > Class**.
  - In the **Name** field, enter `EmployeeBean`. Click **Finish**.
  - Make the `EmployeeBean` class a stateless EJB available for injection by adding the `@Stateless` annotation at the class level:

```
import javax.ejb.Stateless;

@Stateless
public class EmployeeBean {
```

- Inject an instance of the `EmployeeLogger` EJB to use in the method stubs:

```
import javax.ejb.Stateless;
import javax.inject.Inject;
import com.redhat.training.util.EmployeeLogger;

@Stateless
public class EmployeeBean {

 @Inject
 private EmployeeLogger logger;
```

- 2.5. Implement the `createEmployee(Employee e)` method stub, which calls the `logAction` method with an operation value of Create:

```
import javax.ejb.Stateless;
import javax.inject.Inject;
import com.redhat.training.util.EmployeeLogger;
import com.redhat.training.model.Employee;
import com.redhat.training.util.EmployeeLogger.Operation;

@Stateless
public class EmployeeBean {

 @Inject
 private EmployeeLogger logger;

 public void createEmployee(Employee e) {
 logger.logAction(e, Operation.Create);
 }
}
```

- 2.6. Implement the `readEmployeeById(Long id)` method stub, which calls the `logAction` method with an operation value of Read:

```
...
@Stateless
public class EmployeeBean {

 @Inject
 private EmployeeLogger logger;

 public void createEmployee(Employee e) {
 logger.logAction(e, Operation.Create);
 }

 public Employee readEmployeeById(Long id) {

 Employee sample = new Employee();
 sample.setId(id);
 sample.setName("Example Employee");
 logger.logAction(sample, Operation.Read);
 return sample;
 }
}
```

- 2.7. Implement the `updateEmployee(Employee e)` method stub, which calls the `logAction` method with an operation value of Update:

```
...
@Stateless
public class EmployeeBean {

 @Inject
 private EmployeeLogger logger;
```

```

public void createEmployee(Employee e) {
 logger.logAction(e, Operation.Create);
}

public Employee readEmployeeById(Long id) {

 Employee sample = new Employee();
 sample.setId(id);
 sample.setName("Example Employee");
 logger.logAction(sample, Operation.Read);
 return sample;
}
}

public void updateEmployee(Employee e) {
 logger.logAction(e, Operation.Update);
}

```

- 2.8. Implement the `deleteEmployee(Employee e)` method stub, which calls the `logAction` method with an operation value of `Delete`:

```

...
@Stateless
public class EmployeeBean {

 @Inject
 private EmployeeLogger logger;

 public void createEmployee(Employee e) {
 logger.logAction(e, Operation.Create);
 }

 public Employee readEmployeeById(Long id) {

 Employee sample = new Employee();
 sample.setId(id);
 sample.setName("Example Employee");
 logger.logAction(sample, Operation.Read);
 return sample;
 }
}

public void updateEmployee(Employee e) {
 logger.logAction(e, Operation.Update);
}

public void deleteEmployee(Employee e) {
 logger.logAction(e, Operation.Delete);
}

```

- 2.9. Save the changes to the file using `Ctrl+S`.

3. Activate JAX-RS by creating a new application class, and set the application path to `/api`.

- 3.1. Right-click `com.redhat.training.rest`, and click `New > Class`.

- 3.2. In the `Name` field, enter `Service`. Click `Finish`.

- 3.3. Make the new Service class extend the javax.ws.rs.core.Application super class:

```
import javax.ws.rs.core.Application;
public class Service extends Application {
}
```

- 3.4. Set the application path to /api using the @ApplicationPath class-level annotation:

```
import javax.ws.rs.core.Application;
import javax.ws.rs.ApplicationPath;
@ApplicationPath("/api")
public class Service extends Application {
}
```

- 3.5. Save the changes to the file using Ctrl+S.

- 4.** Create the new REST service that injects the EJB and provides three API methods.

- 4.1. Right-click com.redhat.training.rest, and click New > Class.
- 4.2. In the Name field, enter EmployeeRestService. Click Finish.
- 4.3. Make the EmployeeRestService class a stateless EJB:

```
import javax.ejb.Stateless;
@Stateless
public class EmployeeRestService {
```

- 4.4. Add the necessary class-level annotations to set the relative path of this service to employees and denote that it both produces and consumes JSON:

```
import javax.ejb.Stateless;
import javax.ws.rs.Consumes;
import javax.ws.rs.Produces;
import javax.ws.rs.Path;
import javax.ws.rs.core.MediaType;

@Stateless
@Path("employees")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class EmployeeRestService {
```

- 4.5. Inject the EmployeeBean so that it can be used by the REST methods:

```

import javax.ejb.Stateless;
import javax.ws.rs.Consumes;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import javax.inject.Inject;
import com.redhat.training.ejb.EmployeeBean;

@Stateless
@Path("employees")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class EmployeeRestService {

 @Inject
 private EmployeeBean employeeBean;

```

- 4.6. Implement the `getEmployee(Long id)` method, which is mapped to the HTTP GET method and uses a path parameter to map the ID value. The method calls the `readEmployeeById` method on the `EmployeeBean` EJB directly:

```

import javax.ejb.Stateless;
import javax.ws.rs.Consumes;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import javax.inject.Inject;
import com.redhat.training.ejb.EmployeeBean;
import com.redhat.training.model.Employee;
import javax.ws.rs.GET;
import javax.ws.rs.PathParam;

@Stateless
@Path("employees")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class EmployeeRestService {

 @Inject
 private EmployeeBean employeeBean;

 @GET
 @Path("{id}")
 @Produces(MediaType.APPLICATION_JSON)
 public Employee getEmployee(@PathParam("id") Long id) {
 return employeeBean.readEmployeeById(id);
 }

```

- 4.7. Implement the `deleteEmployee(Long id)` method, which is mapped to the HTTP DELETE method and uses a path parameter to map the ID value. The method calls the `deleteEmployee` method on the `EmployeeBean` EJB directly:

```

import javax.ejb.Stateless;
import javax.ws.rs.Consumes;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import javax.inject.Inject;
import com.redhat.training.ejb.EmployeeBean;
import com.redhat.training.model.Employee;
import javax.ws.rs.GET;
import javax.ws.rs.PathParam;
import javax.ws.rs.DELETE;
import javax.ws.rs.PUT;

@Stateless
@Path("employees")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class EmployeeRestService {

 @Inject
 private EmployeeBean employeeBean;

 @GET
 @Path("{id}")
 @Produces(MediaType.APPLICATION_JSON)
 public Employee getEmployee(@PathParam("id") Long id) {
 return employeeBean.readEmployeeById(id);
 }

 @DELETE
 @Path("{id}")
 public void deletePerson(@PathParam("id") Long id) {
 Employee toBeDeleted = employeeBean.readEmployeeById(id);
 employeeBean.deleteEmployee(toBeDeleted);
 }
}

```

- 4.8. Implement the method signature of `saveEmployee(Employee employee)` method, which is mapped to the HTTP POST method. This method consumes JSON data of an employee record and returns a Response object.

```

import javax.ejb.Stateless;
import javax.ws.rs.Consumes;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import javax.inject.Inject;
import com.redhat.training.ejb.EmployeeBean;
import com.redhat.training.model.Employee;
import javax.ws.rs.GET;
import javax.ws.rs.PathParam;
import javax.ws.rs.DELETE;
import javax.ws.rs.PUT;
import javax.ws.rs.core.Response;

```

```

@Stateless
@Path("employees")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class EmployeeRestService {

 @Inject
 private EmployeeBean employeeBean;

 @GET
 @Path("{id}")
 @Produces(MediaType.APPLICATION_JSON)
 public Employee getEmployee(@PathParam("id") Long id) {
 return employeeBean.readEmployeeById(id);
 }

 @DELETE
 @Path("{id}")
 public void deletePerson(@PathParam("id") Long id) {
 Employee toBeDeleted = employeeBean.readEmployeeById(id);
 employeeBean.deleteEmployee(toBeDeleted);
 }

 @POST
 @Consumes(MediaType.APPLICATION_JSON)
 @Produces(MediaType.APPLICATION_JSON)
 public Response savePerson(Employee employee) {

 }
}

```

- 4.9. Copy the contents of the /home/student/JB183/labs/jaxrs-review/saveEmployee.txt and paste it into the method contents. Add an import for the ResponseBuilder:

```

import javax.ws.rs.core.Response.ResponseBuilder;
...Imports omitted...

@Stateless
@Path("employees")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class EmployeeRestService {

 @Inject
 private EmployeeBean employeeBean;

 ...Methods omitted...

 @POST
 @Consumes(MediaType.APPLICATION_JSON)
 @Produces(MediaType.APPLICATION_JSON)
 public Response savePerson(Employee employee) {

```

```

ResponseBuilder builder;
if (employee.getId() == null) {

 Employee newEmployee = new Employee();
 newEmployee.setName(employee.getName());

 employeeBean.createEmployee(newEmployee);

 builder = Response.ok();
} else {

 Employee employeeToUpdate = employeeBean.readEmployeeById(employee.getId());

 if (employeeToUpdate == null) {
 builder = Response.serverError();
 } else {
 employeeBean.updateEmployee(employee);
 builder = Response.ok();
 }
}

return builder.build();
}

```

4.10. Save the changes to the file using **Ctrl+S**.

5. Start EAP by selecting the **Servers** tab in the bottom pane of JBDS. Right-click the server **Red Hat JBoss EAP 7.0 [Stopped]** and click on the green start button to start the server.
6. Deploy the **jaxrs-review** application using the following commands in the terminal window:

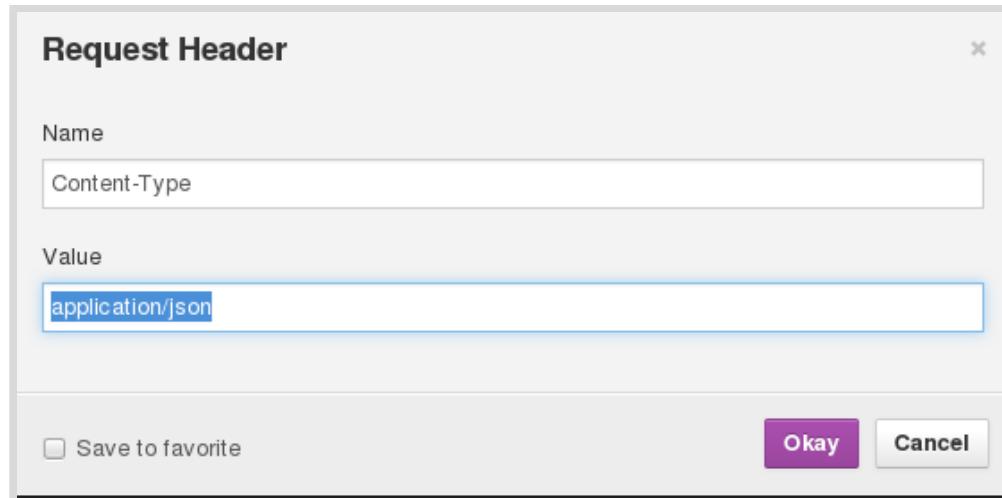
```
[student@workstation ~]$ cd /home/student/JB183/labs/jaxrs-review
[student@workstation jaxrs-review]$ mvn wildfly:deploy
```

7. Read an Employee record using the Firefox REST client plug-in by making a GET request to <http://localhost:8080/jaxrs-review/api/employees>
- 7.1. Start Firefox on the **workstation** VM and click the REST Client plugin icon in the browser's toolbar.



**Figure 10.1: The Firefox REST client plug-in**

- 7.2. In the top toolbar, click **Headers** and click **Custom Header** to add a new custom header to the request.
- 7.3. In the custom header dialog, enter the following information:
  - Name: **Content-Type**
  - Value: **application/json**



**Figure 10.2: Creating a custom request header in the REST Client**

Click Okay.

- 7.4. Select **GET** as the **Method**. In the URL form, enter `http://localhost:8080/jaxrs-review/api/employees/1`.
- 7.5. Verify in the **Response Headers** tab that the **Status Code** is `200 OK`.

Response Headers	Response Body (Raw)	Response Body (Highlight)
<pre> 1. Status Code      : 200 OK 2. Connection       : keep-alive 3. Content-Length   : 0 4. Date             : Fri, 20 Oct 2017 20:27:00 GMT 5. Server            : JBoss-EAP/7 6. X-Powered-By     : Undertow/1 </pre>		

**Figure 10.3: The expected response header**

- 7.6. Check `/home/student/JB183/labs/jaxrs-review/EmployeeLog.txt` and make sure you see the following message:

```
Read Employee: Employee [id=1, name=Example Employee]
```

8. Delete an **Employee** record using the Firefox REST client plugin by making a **DELETE** request to `http://localhost:8080/jaxrs-review/api/employees/1`.
  - 8.1. Select **DELETE** as the **Method**. In the URL form, enter `http://localhost:8080/jaxrs-review/api/employees/1`.
  - 8.2. Click **Send**.

- 8.3. Verify in the **Response Headers** tab that the **Status Code** is **204 No Content**. This is expected because the `deleteEmployee` method has a `void` return type, so there is no content returned to the REST client.
- 8.4. Check `/home/student/JB183/labs/jaxrs-review/EmployeeLog.txt` and make sure you see the following message:

```
Read Employee: Employee [id=1, name=Example Employee]
Delete Employee: Employee [id=1, name=Example Employee]
```

9. Update an **Employee** record using the Firefox REST client plugin by making a POST request to `http://localhost:8080/jaxrs-review/api/employees`, passing in JSON data of an employee record with an ID in the request body.
  - 9.1. Select **POST** as the **Method**. In the URL form, enter `http://localhost:8080/jaxrs-review/api/employees/`.
  - 9.2. In the **Body** section of the request, add the following JSON representation of an **Employee** entity:

```
{"id":1, "name":"Example Employee2"}
```

Click **Send**.

- 9.3. Verify in the **Response Headers** tab that the **Status Code** is **200 OK**.
- 9.4. Check `/home/student/JB183/labs/jaxrs-review/EmployeeLog.txt` and make sure you see the following message:

```
Read Employee: Employee [id=1, name=Example Employee2]
Update Employee: Employee [id=1, name=Example Employee2]
```

10. Create an **Employee** record using the Firefox REST client plugin by making a POST request to `http://localhost:8080/jaxrs-review/api/employees`, passing in JSON data of an employee record **without** an ID in the request body.
  - 10.1. Select **POST** as the **Method**. In the URL form, enter `http://localhost:8080/jaxrs-review/api/employees/`.
  - 10.2. In the **Body** section of the request, add the following JSON representation of an **Employee** entity:

```
{"name":"Example Employee3"}
```

Click **Send**.

- 10.3. Verify in the **Response Headers** tab that the **Status Code** is **200 OK**.
- 10.4. Check `/home/student/JB183/labs/jaxrs-review/EmployeeLog.txt` and make sure you see the following message:

```
Create Employee: Employee [id=null, name=Example Employee3]
```

## Evaluation

As the student user on workstation, run the `lab jaxrs-review grade` argument, to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab jaxrs-review grade
```

After the grading succeeds, undeploy the project, stop the EAP server and close the project in JBDS.

This concludes the lab.

## ► Lab

# Persisting Data with JPA

In this review, you will update the REST API built in the previous comprehensive review to include persistence.

## Outcomes

You should be able to:

- Use JPA to map an entity to a database table.
- Use JPA to map relationships between entities.
- Use a named query to retrieve records from the database.

## Before You Begin

Prepare your computers for this exercise by logging in to `workstation` as `student`, and running the following command:

```
[student@workstation ~]$ lab jpa-review setup
```

## Instructions

1. Open JBDS, and import the `jpa-review` project skeleton. Review the existing code and notice the following:
  - The `src/main/resources/META-INF/persistence.xml` file defines a persistence context that connects to the MySQL datasource defined on the EAP server.
  - Two new model objects have been added in the `com.redhat.training.model` package: `Manager` and `Department`. Each `Employee` has a `Department` and each `Department` has a `Manager`.
2. Create a utility class in the `com.redhat.training.util` package that produces instances of `EntityManager` that use the default persistence context.
3. Update the `Employee` entity to be managed by JPA, and map the relationships between the `Employee`, `Department`, and `Manager` entities using JPA annotations. Make sure to meet the following requirements:
  - The `id` field is the unique identifier for each record and it is automatically generated by the database when a new record is persisted.
  - The `Employee` table in the database uses the `departmentID` column to store the foreign key to the `Department` table.
  - The `Manager` table in the database uses the `departmentID` column to store the foreign key to the `Department` table.
  - The `Department` entity maps its relationships to both `Employee` and `Manager` as the child object in the relationship.

- The `Department` entity eagerly fetches the list of `Employee` instances to which it is related.
- The `Employee` entity includes a JPQL named query, called `findAllForManager`, that takes the ID of a manager record as its parameter. This query pulls the list of employees reporting to that manager. This requires joining to the `department` record then joining to the `manager` record.

**Hint: The JPQL statement that checks the manager ID of an employee is `employee.department.manager.id`**

- The `findAllForManager` query returns `Employee` objects, and uses a named parameter called `managerId` to represent the manager's ID.
4. Update the `EmployeeBean` to support persistence using the `EntityManager`. Make sure to meet the following requirements:
    - The CRUD EJB methods map to the following entity manager methods:
      - `createEmployee: persist(Employee e)`
      - `readEmployeeById: find(Class, Object)`
      - `updateEmployee: merge(Employee e)`
      - `deleteEmployee: remove(Employee e)`
    - The `findAllForManager` method uses the corresponding `findAllForManager` named query. Also, the `managerID` parameter of the query is set to the ID of the `manager` that was passed into the method.
  5. Update the `EmployeeRestService` to support three new methods, and ensure that they meet the following requirements:
    - `getEmployeesForManager(Manager m):`
      - Maps to HTTP GET requests.
      - Uses the relative path `/getByManager/{managerId}`.
      - The `managerId` is a path parameter mapped to the method parameter.
      - Produces XML data.
      - Uses the `findById` method on the `ManagerBean` to look up the manager by its ID value.
      - Takes the manager object that was found, and passes it to the `findAllForManager` method on the `EmployeeBean` and returns the result.
    - `assignEmployee(Long employeeId, Long departmentId):`
      - Maps to HTTP POST requests.
      - Uses the relative path `/assignEmployee/{employeeId}/{departmentId}`.
      - Both the `employeeId` and the `departmentId` are path parameters that are mapped to the method parameters.

- Uses the EmployeeBean to find the employee by ID, then uses the DepartmentBean to find the department by ID. Finally, updates the department record on the employee object, and then calls the updateEmployee method on the EmployeeBean to persist the changes.
- assignManager(Long managerId, Long departmentId)
  - Maps to HTTP POST requests.
  - Uses the relative path /assignManager/{managerId}/{departmentId}.
  - Both the managerId and the departmentId are path parameters mapped to the method parameters.
  - Uses the ManagerBean to find the manager by ID, then uses the DepartmentBean to find the department by ID. Finally, updates the department record on the manager object, and then calls the updateManager method on the ManagerBean to persist the changes.

6. Start JBoss EAP, and use Maven to deploy the jpa-review application.
7. Test the REST service and the three new methods using the Firefox REST Client plugin.

*If you would like to reset the database during testing, run the provided /home/student/JB183/labs/jpa-review/reset-database.sh script.*

- Specify a custom header with the name Content-Type and the value application/json.
- Use an HTTP GET to send a request to the http://localhost:8080/jpa-review/api/employees/getByManager/1 endpoint, to test the getEmployeesForManager method and retrieve the employees for the manager with an id value of 1.

Check the **ResponseBody** to see the XML data for the employees in the department managed by Bob, which is the Sales department:

```
<collection>
<employee>
<department>
<id>1</id>
<manager>
<id>1</id>
<name>Bob</name>
</manager>
<name>Sales</name>
</department>
<id>1</id>
<name>William</name>
</employee>
<employee>
<department>
<id>1</id>
<manager>
<id>1</id>
<name>Bob</name>
```

```

</manager>
<name>Sales</name>
</department>
<id>2</id>
<name>Rose</name>
</employee>
<employee>
<department>
<id>1</id>
<manager>
<id>1</id>
<name>Bob</name>
</manager>
<name>Sales</name>
</department>
<id>3</id>
<name>Pat</name>
</employee>
</collection>

```

- Use an HTTP POST to send a request to the `http://localhost:8080/jpa-review/api/employees/assignEmployee/1/2` endpoint to assign the employee with an ID value of 1 to the department with an ID value of 2.
- Use an HTTP POST to send a request to the `http://localhost:8080/jpa-review/api/employees/assignManager/2/1` to assign a new Manager to the sales department.
- Use an HTTP POST to send a request to the `http://localhost:8080/jpa-review/api/employees/assignManager/1/2` to assign Bob to a new department with an ID value of 2.
- Use an HTTP GET to send a request to the `http://localhost:8080/jpa-review/api/employees/getByManager/1` endpoint, to test the `getEmployeesForManager` method again.

Check the `ResponseBody` to see the XML data for the employees in the department managed by Bob, which is now the Marketing department. This should include the newly assigned employee for a total of 4 employees:

```

<collection>
<employee>
<department>
<id>2</id>
<manager>
<id>1</id>
<name>Bob</name>
</manager>
<name>Marketing</name>
</department>
<id>1</id>
<name>William</name>
</employee>
<employee>
<department>

```

```
<id>2</id>
<manager>
 <id>1</id>
 <name>Bob</name>
</manager>
<name>Marketing</name>
</department>
<id>4</id>
<name>Rodney</name>
</employee>
<employee>
 <department>
 <id>2</id>
 <manager>
 <id>1</id>
 <name>Bob</name>
 </manager>
 <name>Marketing</name>
 </department>
<id>5</id>
<name>Kim</name>
</employee>
<employee>
 <department>
 <id>2</id>
 <manager>
 <id>1</id>
 <name>Bob</name>
 </manager>
 <name>Marketing</name>
 </department>
<id>6</id>
<name>Tom</name>
</employee>
</collection>
```

## Steps

### Evaluation

As the **student** user on **workstation**, run the **lab jpa-review** script with the **grade** argument, to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab jpa-review grade
```

After the grading succeeds, stop the EAP server and close the project in JBDS.

This concludes the lab.

## ► Solution

# Persisting Data with JPA

In this review, you will update the REST API built in the previous comprehensive review to include persistence.

## Outcomes

You should be able to:

- Use JPA to map an entity to a database table.
- Use JPA to map relationships between entities.
- Use a named query to retrieve records from the database.

## Before You Begin

Prepare your computers for this exercise by logging in to `workstation` as `student`, and running the following command:

```
[student@workstation ~]$ lab jpa-review setup
```

## Instructions

1. Open JBDS, and import the `jpa-review` project skeleton. Review the existing code and notice the following:
  - The `src/main/resources/META-INF/persistence.xml` file defines a persistence context that connects to the MySQL datasource defined on the EAP server.
  - Two new model objects have been added in the `com.redhat.training.model` package: `Manager` and `Department`. Each `Employee` has a `Department` and each `Department` has a `Manager`.
2. Create a utility class in the `com.redhat.training.util` package that produces instances of `EntityManager` that use the default persistence context.
3. Update the `Employee` entity to be managed by JPA, and map the relationships between the `Employee`, `Department`, and `Manager` entities using JPA annotations. Make sure to meet the following requirements:
  - The `id` field is the unique identifier for each record and it is automatically generated by the database when a new record is persisted.
  - The `Employee` table in the database uses the `departmentID` column to store the foreign key to the `Department` table.
  - The `Manager` table in the database uses the `departmentID` column to store the foreign key to the `Department` table.
  - The `Department` entity maps its relationships to both `Employee` and `Manager` as the child object in the relationship.

- The `Department` entity eagerly fetches the list of `Employee` instances to which it is related.
- The `Employee` entity includes a JPQL named query, called `findAllForManager`, that takes the ID of a manager record as its parameter. This query pulls the list of employees reporting to that manager. This requires joining to the `department` record then joining to the `manager` record.

**Hint: The JPQL statement that checks the manager ID of an employee is `employee.department.manager.id`**

- The `findAllForManager` query returns `Employee` objects, and uses a named parameter called `managerId` to represent the manager's ID.
4. Update the `EmployeeBean` to support persistence using the `EntityManager`. Make sure to meet the following requirements:
    - The CRUD EJB methods map to the following entity manager methods:
      - `createEmployee: persist(Employee e)`
      - `readEmployeeById: find(Class, Object)`
      - `updateEmployee: merge(Employee e)`
      - `deleteEmployee: remove(Employee e)`
    - The `findAllForManager` method uses the corresponding `findAllForManager` named query. Also, the `managerID` parameter of the query is set to the ID of the manager that was passed into the method.
  5. Update the `EmployeeRestService` to support three new methods, and ensure that they meet the following requirements:
    - `getEmployeesForManager(Manager m):`
      - Maps to HTTP GET requests.
      - Uses the relative path `/getByManager/{managerId}`.
      - The `managerId` is a path parameter mapped to the method parameter.
      - Produces XML data.
      - Uses the `findById` method on the `ManagerBean` to look up the manager by its ID value.
      - Takes the manager object that was found, and passes it to the `findAllForManager` method on the `EmployeeBean` and returns the result.
    - `assignEmployee(Long employeeId, Long departmentId):`
      - Maps to HTTP POST requests.
      - Uses the relative path `/assignEmployee/{employeeId}/{departmentId}`.
      - Both the `employeeId` and the `departmentId` are path parameters that are mapped to the method parameters.

- Uses the `EmployeeBean` to find the employee by ID, then uses the `DepartmentBean` to find the department by ID. Finally, updates the department record on the employee object, and then calls the `updateEmployee` method on the `EmployeeBean` to persist the changes.
  - `assignManager(Long managerId, Long departmentId)`
    - Maps to HTTP POST requests.
    - Uses the relative path `/assignManager/{managerId}/{departmentId}`.
    - Both the `managerId` and the `departmentId` are path parameters mapped to the method parameters.
    - Uses the `ManagerBean` to find the manager by ID, then uses the `DepartmentBean` to find the department by ID. Finally, updates the department record on the manager object, and then calls the `updateManager` method on the `ManagerBean` to persist the changes.
6. Start JBoss EAP, and use Maven to deploy the `jpa-review` application.
  7. Test the REST service and the three new methods using the Firefox REST Client plugin.
- If you would like to reset the database during testing, run the provided `/home/student/JB183/labs/jpa-review/reset-database.sh` script.*
- Specify a custom header with the name `Content-Type` and the value `application/json`.
  - Use an HTTP GET to send a request to the `http://localhost:8080/jpa-review/api/employees/getByManager/1` endpoint, to test the `getEmployeesForManager` method and retrieve the employees for the manager with an `id` value of 1.

Check the `ResponseBody` to see the XML data for the employees in the department managed by Bob, which is the Sales department:

```
<collection>
<employee>
<department>
<id>1</id>
<manager>
<id>1</id>
<name>Bob</name>
</manager>
<name>Sales</name>
</department>
<id>1</id>
<name>William</name>
</employee>
<employee>
<department>
<id>1</id>
<manager>
<id>1</id>
<name>Bob</name>
```

```

</manager>
<name>Sales</name>
</department>
<id>2</id>
<name>Rose</name>
</employee>
<employee>
<department>
<id>1</id>
<manager>
<id>1</id>
<name>Bob</name>
</manager>
<name>Sales</name>
</department>
<id>3</id>
<name>Pat</name>
</employee>
</collection>

```

- Use an HTTP POST to send a request to the `http://localhost:8080/jpa-review/api/employees/assignEmployee/1/2` endpoint to assign the employee with an ID value of 1 to the department with an ID value of 2.
- Use an HTTP POST to send a request to the `http://localhost:8080/jpa-review/api/employees/assignManager/2/1` to assign a new Manager to the sales department.
- Use an HTTP POST to send a request to the `http://localhost:8080/jpa-review/api/employees/assignManager/1/2` to assign Bob to a new department with an ID value of 2.
- Use an HTTP GET to send a request to the `http://localhost:8080/jpa-review/api/employees/getByManager/1` endpoint, to test the `getEmployeesForManager` method again.

Check the **ResponseBody** to see the XML data for the employees in the department managed by Bob, which is now the **Marketing** department. This should include the newly assigned employee for a total of 4 employees:

```

<collection>
<employee>
<department>
<id>2</id>
<manager>
<id>1</id>
<name>Bob</name>
</manager>
<name>Marketing</name>
</department>
<id>1</id>
<name>William</name>
</employee>
<employee>
<department>

```

```

<id>2</id>
<manager>
 <id>1</id>
 <name>Bob</name>
</manager>
<name>Marketing</name>
</department>
<id>4</id>
<name>Rodney</name>
</employee>
<employee>
 <department>
 <id>2</id>
 <manager>
 <id>1</id>
 <name>Bob</name>
 </manager>
 <name>Marketing</name>
 </department>
<id>5</id>
<name>Kim</name>
</employee>
<employee>
 <department>
 <id>2</id>
 <manager>
 <id>1</id>
 <name>Bob</name>
 </manager>
 <name>Marketing</name>
 </department>
<id>6</id>
<name>Tom</name>
</employee>
</collection>

```

## Steps

### Steps

1. Open JBDS and import the Maven project.
  - 1.1. Open JBDS by double-clicking the JBoss Developer Studio icon on the workstation desktop. Set the workspace to /home/student/JB183/workspace and click OK.
  - 1.2. In the JBDS menu, click **File > Import** to open the **Import** wizard.
  - 1.3. On the **Select** page, click **Maven > Existing Maven Projects**, and then click **Next**.
  - 1.4. On the **Maven projects** page, click **Browse** to open the **Select root folder** window. Navigate to the /home/student/JB183/labs/ directory. Select the jpa-review folder, and then click **OK**.
  - 1.5. On the **Maven projects** page, click **Finish**.

- 1.6. Watch the JBDS status bar to monitor the progress of the import operation. It may take a few minutes to download all the required dependencies.
2. Add a utility class that produces EntityManager instances by injecting the default persistence context.
  - 2.1. Expand the **jpa-review** item in the **Project Explorer** tab in the left pane of JBDS, and then click on **jpa-review > Java Resources**. Right-click **com.redhat.training.util** and click **New > Class**.
  - 2.2. In the **Name** field, enter **Resources**. Click **Finish**.
  - 2.3. Make the **Resources** class a producer for EntityManager instances using the **@Produces** annotation, and link the entity manager to the default persistence context using the **@PersistenceContext** annotation:

```
import javax.enterprise.inject.Produces;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

public class Resources {

 @Produces
 @PersistenceContext
 private EntityManager em;

}
```

- 2.4. Save your changes to the file using **Ctrl+S**.
3. Update the Employee model class to be a JPA managed entity.
  - 3.1. Open the Employee class by expanding the **jpa-review** item in the **Project Explorer** tab in the left pane of JBDS, and then click on **jpa-review > Java Resources > src/main/java > com.redhat.training.model** to expand it. Double-click the **Employee.java** file.

Add the **@Entity** annotation to mark this class an entity managed by JPA:

```
...
//TODO Make this class an Entity
@Entity
//TODO Add a named query to find all employees for a given manager
public class Employee {
...
```

- 3.2. Configure the **id** field to be the unique identifier for the Employee class using the **@Id** annotation. Mark it as a generated value using the **@GeneratedValue** annotation with a value of **IDENTITY**:

```
...
//TODO Make this class an Entity
@Entity
```

```
//TODO Add a named query to find all employees for a given manager
public class Employee {

 //TODO mark this as the Id field
 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;
```

3.3. Save your changes to the file using **Ctrl+S**.

**4.** Map the relationships between the `Employee`, `Manager`, and `Department` entities.

4.1. In the `Employee` class, map the many-to-one relationship with the `Department` entity.

Use the `@ManyToOne` annotation to tell JPA to map the relationship and the `@JoinColumn` annotation to specify the `departmentID` column as the foreign key to the `Department` table:

```
...
//TODO Make this class an Entity
@Entity
//TODO Add a named query to find all employees for a given manager
public class Employee {

 //TODO mark this as the Id field
 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Long id;

 private String name;

 //TODO map the relationship using JPA annotations
 @ManyToOne
 @JoinColumn(name="departmentID")
 private Department department;
```

4.2. Save your changes to the file using **Ctrl+S**.

4.3. Open the `Department` class by expanding the `jpa-review` item in the **Project Explorer** tab in the left pane of JBDS. Click on `jpa-review > Java Resources > src/main/java > com.redhat.training.model` to expand it. Double-click the `Department.java` file.

4.4. Map the one-to-many relationship with the `Employee` entity, which is mapped by the `department` field. Load the related `Employee` entities eagerly.

Use the `@OneToMany` annotation and the `mappedBy` attribute. Additionally, set the fetch type to `EAGER`:

```
@Entity
public class Department {

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```

private Long id;

private String name;

//TODO map the relationship using JPA annotations

private Manager manager;

//TODO map the relationship using JPA annotations, fetch the list eagerly
@OneToMany(mappedBy="department", fetch=FetchType.EAGER)
private Set<Employee> employees;

```

- 4.5. Map the one-to-one relationship with the `Manager` entity, which is mapped by the `department` field.

Use the `@OneToOne` annotation and the `mappedBy` attribute:

```

@Entity
public class Department {

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;

private String name;

//TODO map the relationship using JPA annotations
@OneToOne(mappedBy="department")
private Manager manager;

//TODO map the relationship using JPA annotations, fetch the list eagerly
@OneToMany(mappedBy="department", fetch=FetchType.EAGER)
private Set<Employee> employees;

```



### Note

The error on the `OneToOne` annotation is resolved in the next step.

- 4.6. Save your changes to the file using `Ctrl+S`.
- 4.7. Open the `Manager` class by expanding the `jpa-review` item in the Project Explorer tab in the left pane of JBDS. Click on `jpa-review > Java Resources > src/main/java > com.redhat.training.model` to expand it. Double-click the `Manager.java` file.
- 4.8. Map the one-to-one relationship with the `Department` entity, which is related using the `departmentID` column as a foreign key.
- Use the `@OneToOne` annotation to tell JPA to map the relationship. Use the `@JoinColumn` annotation to specify the `departmentID` column as the foreign key to the `Department` table:

```

@Entity
public class Manager {

```

```

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;

private String name;

//TODO map the relationship using JPA annotations
@OneToOne
@JoinColumn(name="departmentID")
private Department department;

```

4.9. Save your changes to the file using **Ctrl+S**.

5. Add a named query to find all the employees that report to a given manager, and then update the **EmployeeBean** to use the named query.
  - 5.1. Open the **Employee** class by expanding the **jpa-review** item in the **Project Explorer** tab in the left pane of JBDS. Click on **jpa-review > Java Resources > src/main/java > com.redhat.training.model** to expand it. Double-click the **Employee.java** file.  
Add the **@NamedQuery** annotation to add register the query with JPA. Name the query **findAllForManager**, and use a named parameter called **:managerId**:

```

...
//TODO Make this class an Entity
@Entity
//TODO Add a named query to find all employees for a given manager
@NamedQuery(name="findAllForManager", query="select e from Employee e where
e.department.manager.id = :managerId")
public class Employee {

```

5.2. Save your changes to the file using **Ctrl+S**.

6. Implement the persistence functionality in the **EmployeeBean** by injecting an **EntityManager** and calling the appropriate methods for each EJB method.
  - 6.1. Open the **EmployeeBean** class by expanding the **jpa-review** item in the **Project Explorer** tab in the left pane of JBDS. Click on **jpa-review > Java Resources > src/main/java > com.redhat.training.ejb** to expand it. Double-click the **EmployeeBean.java** file.
  - 6.2. Inject an instance of the **EntityManager** class into the **EmployeeBean** using the **@Inject** annotation:

```

@Stateless
public class EmployeeBean {

 //TODO inject EntityManager
 @Inject
 private EntityManager em;

```

- 6.3. Implement the **createEmployee** persistence functionality using the **persist** method on the entity manager:

```
public void createEmployee(Employee e) {
 //TODO persist employee
 em.persist(e);
 logger.logAction(e, Operation.Create);
}
```

- 6.4. Implement the `readEmployeeById` persistence functionality by updating the method to use the `find` method on the entity manager:

```
public Employee readEmployeeById(Long id) {
 //TODO find the employee by its ID
 Employee employee = em.find(Employee.class, id)
 logger.logAction(employee, Operation.Read);
 return employee;
}
```

- 6.5. Implement the `updateEmployee` persistence functionality by updating the method to use the `merge` method on the entity manager:

```
public void updateEmployee(Employee e) {
 //TODO merge the employee record
 em.merge(e);
 logger.logAction(e, Operation.Update);
}
```

- 6.6. Implement the `deleteEmployee` persistence functionality by updating the method to use the `remove` method on the entity manager:

```
public void deleteEmployee(Employee e) {
 //TODO remove the employee record
 em.remove(e);
 logger.logAction(e, Operation.Delete);
}
```

- 6.7. Implement the `findAllForManager` persistence functionality by updating the method to create a `TypedQuery` using the named query created in the previous step. Set the ID of the manager object that was passed in as the `managerId` query parameter. Update the return statement to return the result:

```
public List<Employee> findAllForManager(Manager manager) {
 //TODO use the named query to find all the employees for a manager
 TypedQuery<Employee> query =
 em.createNamedQuery("findAllForManager", Employee.class);
 query.setParameter("managerId", manager.getId());
 return query.getResultList();
}
```

- 6.8. Save your changes to the file using `Ctrl+S`.

7. Update the `EmployeeRestService` to add a new REST method named `getEmployeesForManager` that uses the `findAllForManager` method on the `EmployeeBean` to retrieve all employees for a department.
- 7.1. Open the `EmployeeRestService` class by expanding the `jpa-review` item in the `Project Explorer` tab in the left pane of JBDS. Click on `jpa-review > Java Resources > src/main/java > com.redhat.training.rest` to expand it. Double-click the `EmployeeRestService.java` file.
  - 7.2. Add the new method after the comment `//TODO add method to pull an XML list of employees for a given manager`:

```
//TODO add method to pull an XML list of employees for a given manager id
public List<Employee> getEmployeesForManager(Long managerId){
 Manager manager = managerBean.findById(managerId);
 return employeeBean.findAllForManager(manager);
}
```

- 7.3. Associate the method with HTTP GET requests using the `@GET` annotation:

```
@GET
public List<Employee> getEmployeesForManager(Long managerId){
 Manager manager = managerBean.findById(managerId);
 return employeeBean.findAllForManager(manager);
}
```

- 7.4. Set the relative path of the method to `/ getByManager/{managerId}` using the `@Path` annotation:

```
@GET
@Path("getByManager/{managerId}")
public List<Employee> getEmployeesForManager(Long managerId){
 Manager manager = managerBean.findById(managerId);
 return employeeBean.findAllForManager(manager);
}
```

- 7.5. Add the `@PathParam` annotation on the `managerId` method parameter to map the path parameter from the URL into the method automatically:

```
@GET
@Path("getByManager/{managerId}")
public List<Employee> getEmployeesForManager(@PathParam("managerId") Long
managerId){
 Manager manager = managerBean.findById(managerId);
 return employeeBean.findAllForManager(manager);
}
```

- 7.6. Specify that the method produces XML data using the `@Produces` annotation:

```

@GET
@Path("getByManager/{managerId}")
@Produces(MediaType.APPLICATION_XML)
public List<Employee> getEmployeesForManager(@PathParam("managerId") Long
managerId){
 Manager manager = managerBean.findById(managerId);
 return employeeBean.findAllForManager(manager);
}

```

7.7. Save your changes to the file using **Ctrl+S**.

8. Update the `EmployeeRestService` to add a new REST method named `assignEmployee` that assigns an employee to a department using IDs.

8.1. Add a new method after the comment `//TODO add REST method to assign an employee to a department by ID`. Use the `employeeBean` to look up the employee by its ID, and the `departmentBean` to look up the department. Then, update the department record on the employee and use the `updateEmployee` method to persist the changes:

```

//TODO add REST method to assign an employee to a department by ID
public void assignEmployee(Long employeeId, Long departmentId) {
 Employee e = employeeBean.readEmployeeById(employeeId);
 Department d = departmentBean.findById(departmentId);
 e.setDepartment(d);
 employeeBean.updateEmployee(e);
}

```

8.2. Associate the method with HTTP POST requests using the `@POST` annotation:

```

//TODO add REST method to assign an employee to a department by ID
@POST
public void assignEmployee(Long employeeId, Long departmentId) {
 Employee e = employeeBean.readEmployeeById(employeeId);
 Department d = departmentBean.findById(departmentId);
 e.setDepartment(d);
 employeeBean.updateEmployee(e);
}

```

8.3. Set the relative path of the method to `/assignEmployee/{employeeId}/{departmentId}` using the `@Path` annotation:

```

@POST
@Path("assignEmployee/{employeeId}/{departmentId}")
public void assignEmployee(Long employeeId, Long departmentId) {
 Employee e = employeeBean.readEmployeeById(employeeId);
 Department d = departmentBean.findById(departmentId);
 e.setDepartment(d);
 employeeBean.updateEmployee(e);
}

```

- 8.4. Add the `@PathParam` annotations on the method parameters to map them from the URL path:

```
@POST
@Path("assignEmployee/{employeeId}/{departmentId}")
public void assignEmployee(@PathParam("employeeId") Long
 employeeId, @PathParam("departmentId") Long departmentId) {
 Employee e = employeeBean.readEmployeeById(employeeId);
 Department d = departmentBean.findById(departmentId);
 e.setDepartment(d);
 employeeBean.updateEmployee(e);
}
```

- 8.5. Save your changes to the file using **Ctrl+S**.

9. Update the `EmployeeRestService` to add a new REST method named `assignManager` that assigns a manager to a department using IDs.

- 9.1. Add a new method after the comment `//TODO add REST method to assign an manager to a department by ID`. Use the `managerBean` to look up the manager by its ID, and the `departmentBean` to look up the department. Then update the department record on the manager and use the `updateManager` method to persist the changes:

```
//TODO add REST method to assign a manager to a department by ID
public void assignManager(Long managerId, Long departmentId) {
 Manager m = managerBean.findById(managerId);
 Department d = departmentBean.findById(departmentId);
 m.setDepartment(d);
 managerBean.updateManager(m);
}
```

- 9.2. Associate the method with HTTP POST requests using the `@POST` annotation:

```
//TODO add REST method to assign an employee to a department by ID
@POST
public void assignManager(Long managerId, Long departmentId) {
 Manager m = managerBean.findById(managerId);
 Department d = departmentBean.findById(departmentId);
 m.setDepartment(d);
 managerBean.updateManager(m);
}
```

- 9.3. Set the relative path of the method to `/assignEmployee/{managerId}/{departmentId}` using the `@Path` annotation:

```

@POST
@Path("assignManager/{managerId}/{departmentId}")
public void assignEmployee(Long employeeId, Long departmentId) {
 Manager m = managerBean.findById(managerId);
 Department d = departmentBean.findById(departmentId);
 m.setDepartment(d);
 managerBean.updateManager(m);
}

```

- 9.4. Add the `@PathParam` annotations on the method parameters to map them from the URL path:

```

@POST
@Path("assignManager/{managerId}/{departmentId}")
public void assignManager(@PathParam("managerId") Long
 managerId, @PathParam("departmentId") Long departmentId) {
 Manager m = managerBean.findById(managerId);
 Department d = departmentBean.findById(departmentId);
 m.setDepartment(d);
 managerBean.updateManager(m);
}

```

- 9.5. Save the changes to the file using **Ctrl+S**.

10. Start EAP by selecting the **Servers** tab in the bottom pane of JBDS. Right-click the server **Red Hat JBoss EAP 7.0 [Stopped]** and click on the green start button to start the server.
11. Deploy the jpa-review application using the following commands in the terminal window:

```

[student@workstation ~]$ cd /home/student/JB183/labs/jpa-review
[student@workstation jpa-review]$ mvn wildfly:deploy

```

12. Test the new REST method to pull employees for a manager using the REST client Firefox plugin.
  - 12.1. Start Firefox on the **workstation** VM and click the REST Client plugin icon in the browser's toolbar.
  - 12.2. In the top toolbar, click **Headers** and click **Custom Header** to add a new custom header to the request.
  - 12.3. In the custom header dialog, enter the following information:
    - Name: **Content-Type**
    - Value: **application/json**
 Click **Okay**.
  - 12.4. Select **GET** as the **Method**. In the URL form, enter `http://localhost:8080/jpa-review/api/employees/getByManager/1`.  
Click **Send**.
  - 12.5. Verify in the **Response Headers** tab that the **Status Code** is **200 OK**.

Check the **Response Body** tab to see the XML data and verify that it matches what is expected:

```
<collection>
 <employee>
 <department>
 <id>1</id>
 <manager>
 <id>1</id>
 <name>Bob</name>
 </manager>
 <name>Sales</name>
 </department>
 <id>1</id>
 <name>William</name>
 </employee>
 <employee>
 <department>
 <id>1</id>
 <manager>
 <id>1</id>
 <name>Bob</name>
 </manager>
 <name>Sales</name>
 </department>
 <id>2</id>
 <name>Rose</name>
 </employee>
 <employee>
 <department>
 <id>1</id>
 <manager>
 <id>1</id>
 <name>Bob</name>
 </manager>
 <name>Sales</name>
 </department>
 <id>3</id>
 <name>Pat</name>
 </employee>
</collection>
```

13. Test the two new REST methods for reassigning managers and employees to new departments.
  - 13.1. Select **POST** as the **Method**. In the URL form, enter `http://localhost:8080/jpa-review/api/employees/assignEmployee/1/2` to reassign employee with an ID value of 1 to a new department with the ID of 2.
  - 13.2. Click **Send**.
  - 13.3. Verify in the **Response Headers** tab that the **Status Code** is `204 No Content`. This is expected because the return type is `void`.

- 13.4. Select **POST** as the **Method**. In the URL form, enter `http://localhost:8080/jaxrs-review/api/employees/assignManager/1/2` to reassign manager with an ID value of 1 to a new department.
- 13.5. Click **Send**.
- 13.6. Verify in the **Response Headers** tab that the **Status Code** is **204 No Content**. This is expected because the return type is **void**.
- 13.7. Select **POST** as the **Method**. In the URL form, enter `http://localhost:8080/jpa-review/api/employees/assignManager/2/1` to reassign the manager with an ID value of 2 to the sales department, replacing Bob.
- 13.8. Click **Send**.
- 13.9. Verify in the **Response Headers** tab that the **Status Code** is **204 No Content**. This is expected because the return type is **void**.
- 13.10. Run the `getByManager` method again to pull a list of Bob's new direct reports.  
Select **GET** as the **Method**. In the URL form, enter `http://localhost:8080/jpa-review/api/employees/getByManager/1`.  
Click **Send**.
- 13.11. Verify in the **Response Headers** tab that the **Status Code** is **200 OK**.  
Check the **Response Body** tab to see the XML data and verify that it matches what is expected:

```
<collection>
<employee>
 <department>
 <id>2</id>
 <manager>
 <id>1</id>
 <name>Bob</name>
 </manager>
 <name>Marketing</name>
 </department>
 <id>1</id>
 <name>William</name>
</employee>
<employee>
 <department>
 <id>2</id>
 <manager>
 <id>1</id>
 <name>Bob</name>
 </manager>
 <name>Marketing</name>
 </department>
 <id>4</id>
 <name>Rodney</name>
</employee>
<employee>
 <department>
 <id>2</id>
```

```
<manager>
<id>1</id>
<name>Bob</name>
</manager>
<name>Marketing</name>
</department>
<id>5</id>
<name>Kim</name>
</employee>
<employee>
<department>
<id>2</id>
<manager>
<id>1</id>
<name>Bob</name>
</manager>
<name>Marketing</name>
</department>
<id>6</id>
<name>Tom</name>
</employee>
</collection>
```

## Evaluation

As the student user on `workstation`, run the `lab jpa-review` script with the `grade` argument, to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab jpa-review grade
```

After the grading succeeds, stop the EAP server and close the project in JBDS.

This concludes the lab.

## ▶ Lab

# Securing the REST API with JAAS

In this review, you will secure the REST API with authentication and authorization using JAAS.

## Outcomes

You should be able to:

- Connect a Java web application to a JBoss EAP security domain using the necessary configuration files.
- Secure specific resources using basic authentication.
- Limit authorization of users that can use specific RESTEasy services using JAAS annotations to configure role-based authentication.

## Before You Begin

Prepare your computers for this exercise by logging in to `workstation` as `student`, and running the following command:

```
[student@workstation ~]$ lab jaas-review setup
```

## Instructions

1. Open JBDS, and import the `jaas-review` project skeleton.
2. Add a new REST service class that meets the following requirements:
  - Available at the following path: `http://localhost:8080/jaas-review/api/healthCheck`
  - Contains the following method and implementation:

```
public Response getHealthCheck() {
 ResponseBuilder builder = Response.status(Status.OK);
 return builder.build();
}
```

- Associated to HTTP GET requests.
  - Takes no arguments, and always returns an HTTP status 200 OK response.
3. Enable basic authentication on the existing `EmployeeRestService`. Make sure to meet the following requirements:
    - Start JBoss EAP and create a new security domain for EAP using the provided `create-sd.sh` script. *Be sure to start JBoss EAP before running this script.* This security domain uses the `UsersRoles` login module to read the provided `/home/student/JB183/labs/jaas-review/todo-users.properties` user property file for authentication and the `/home/student/JB183/labs/jaas-review/todo-roles.properties` file for authorization. The following users are included:

### Comprehensive Review Lab Users

Username	Password	Role
employee	redhat1!	employee
manager	redhat1!	manager
superuser	redhat1!	superuser

- Configure the application to use this security domain.
- Enable role-based authentication for RESTEasy.
- Use a security constraint to apply the new security **only** to the EmployeeRestService, not to the new HealthCheckService, which should be available without any authentication.
- Define the following roles:
  - employee
  - manager
  - superuser
- Configure the application to use BASIC authentication to require a user name and password for all secured resources.
- Use the following table to configure each of the following methods in the EmployeeRestService class to be restricted to proper roles:

### Method to Role Mapping

Method	Roles Allowed
getEmployee(Long id)	All
getEmployeesForManager(Long managerId)	All
assignEmployee(Long employeeId, Long departmentId)	manager, superuser
assignManager(Long managerId, Long departmentId)	superuser

4. Use Maven to deploy the jaas-review application.



#### Note

The data in the database was reset before this comprehensive review lab.

5. Test the unsecured health check REST service method using the Firefox REST Client plugin and no authentication.

- Use an HTTP GET to send a request to the `http://localhost:8080/jaas-review/api/healthCheck` endpoint, to test the `getHealthCheck` method.
6. Test the secured REST service methods using the Firefox REST Client plugin and basic authentication. Make sure that each role is able to access only the correct methods.
- Use the credentials found in the table above, to set authentication onto the HTTP requests sent by the client.
  - Specify a custom header with the name **Content -Type** and the value `application/json`.

## Steps

### Evaluation

As the `student` user on `workstation`, run the `lab jaas-review` script with the `grade` argument to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab jaas-review grade
```

After the grading succeeds, stop the EAP server and close the project in JBDS.

This concludes the lab.

## ► Solution

# Securing the REST API with JAAS

In this review, you will secure the REST API with authentication and authorization using JAAS.

## Outcomes

You should be able to:

- Connect a Java web application to a JBoss EAP security domain using the necessary configuration files.
- Secure specific resources using basic authentication.
- Limit authorization of users that can use specific RESTEasy services using JAAS annotations to configure role-based authentication.

## Before You Begin

Prepare your computers for this exercise by logging in to `workstation` as `student`, and running the following command:

```
[student@workstation ~]$ lab jaas-review setup
```

## Instructions

1. Open JBDS, and import the `jaas-review` project skeleton.
2. Add a new REST service class that meets the following requirements:
  - Available at the following path: `http://localhost:8080/jaas-review/api/healthCheck`
  - Contains the following method and implementation:

```
public Response getHealthCheck() {
 ResponseBuilder builder = Response.status(Status.OK);
 return builder.build();
}
```

3. Associated to HTTP GET requests.
4. Takes no arguments, and always returns an HTTP status 200 OK response.
5. Enable basic authentication on the existing `EmployeeRestService`. Make sure to meet the following requirements:
  - Start JBoss EAP and create a new security domain for EAP using the provided `create-sd.sh` script. Be sure to start JBoss EAP before running this script. This security domain uses the `UsersRoles` login module to read the provided `/home/student/JB183/labs/jaas-review/todo-users.properties` user property file for authentication and the `/home/student/JB183/labs/jaas-review/todo-roles.properties` file for authorization. The following users are included:

### Comprehensive Review Lab Users

Username	Password	Role
employee	redhat1!	employee
manager	redhat1!	manager
superuser	redhat1!	superuser

- Configure the application to use this security domain.
- Enable role-based authentication for RESTEasy.
- Use a security constraint to apply the new security **only** to the `EmployeeRestService`, not to the new `HealthCheckService`, which should be available without any authentication.
- Define the following roles:
  - employee
  - manager
  - superuser
- Configure the application to use BASIC authentication to require a user name and password for all secured resources.
- Use the following table to configure each of the following methods in the `EmployeeRestService` class to be restricted to proper roles:

#### Method to Role Mapping

Method	Roles Allowed
<code>getEmployee(Long id)</code>	All
<code>getEmployeesForManager(Long managerId)</code>	All
<code>assignEmployee(Long employeeId, Long departmentId)</code>	manager, superuser
<code>assignManager(Long managerId, Long departmentId)</code>	superuser

- Use Maven to deploy the `jaas-review` application.



#### Note

The data in the database was reset before this comprehensive review lab.

- Test the unsecured health check REST service method using the Firefox REST Client plugin and no authentication.

- Use an HTTP GET to send a request to the `http://localhost:8080/jaas-review/api/healthCheck` endpoint, to test the `getHealthCheck` method.

6. Test the secured REST service methods using the Firefox REST Client plugin and basic authentication. Make sure that each role is able to access only the correct methods.
  - Use the credentials found in the table above, to set authentication onto the HTTP requests sent by the client.
  - Specify a custom header with the name `Content-Type` and the value `application/json`.

## Steps

### Steps

1. Open JBDS and import the Maven project.
  - 1.1. Open JBDS by double-clicking the JBoss Developer Studio icon on the workstation desktop. Set the workspace to `/home/student/JB183/workspace` and click **OK**.
  - 1.2. In the JBDS menu, click **File > Import** to open the **Import** wizard.
  - 1.3. On the **Select** page, click **Maven > Existing Maven Projects**, and then click **Next**.
  - 1.4. In the **Maven projects** page, click **Browse** to open the **Select root folder** window. Navigate to the `/home/student/JB183/labs/` directory. Select the `jaas-review` folder, and then click **OK**.
  - 1.5. On the **Maven projects** page, click **Finish**.
  - 1.6. Watch the JBDS status bar to monitor the progress of the import operation. It may take a few minutes to download all the required dependencies.
2. Add the new health check service.
  - 2.1. Right-click `com.redhat.training.rest` and click **New > Class**.
  - 2.2. In the **Name** field, enter `HealthCheckService`. Click **Finish**.
  - 2.3. Mark the `HealthCheckService` as a stateless EJB:

```
import javax.ejb.Stateless;

@Stateless
public class HealthCheckService {
```

- 2.4. Configure the relative path of the `HealthCheckService` REST service to be `/healthCheck` using the `@Path` annotation:

```
import javax.ejb.Stateless;
import javax.ws.rs.Path;

@Stateless
@Path("/healthcheck")
public class HealthCheckService {
```

- 2.5. Implement the health check method to return an HTTP status code 200 "OK" response for every request:

```
import javax.ejb.Stateless;
import javax.ws.rs.Path;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.ResponseBuilder;
import javax.ws.rs.core.Response.Status;

@Stateless
@Path("/healthcheck")
public class HealthCheckService {

 public Response getHealthCheck() {

 ResponseBuilder builder = Response.status(Status.OK);
 return builder.build();
 }
}
```



### Note

The `getHealthCheck()` method is available in the `/home/student/JB183/labs/jaas-review/healthCheck.txt` file.

- 2.6. Associate the `getHealthCheck` method with HTTP GET requests using the RESTEasy `@GET` annotation:

```
import javax.ejb.Stateless;
import javax.ws.rs.Path;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.ResponseBuilder;
import javax.ws.rs.core.Response.Status;
import javax.ws.rs.GET;

@Stateless
@Path("/healthcheck")
public class HealthCheckService {

 @GET
 public Response getHealthCheck() {

 ResponseBuilder builder = Response.status(Status.OK);
```

```

 return builder.build();
}
}

```

- 2.7. Save your changes by pressing **Ctrl+S**.
  3. Start JBoss EAP from within JBDS.
- Select the **Servers** tab in JBDS. Right-click on the server entry **Red Hat JBoss EAP 7.0 [Stopped]** and click the green **Start** option to start the server.
4. Use the `/home/student/JB183/labs/jaas-review/create-sd.sh` script to create a `UsersRoles` security domain named `userroles`.
- This security domain uses the `UsersRoles` login module to read the provided `/home/student/JB183/labs/jaas-review/todo-users.properties` user property file for authentication and the `/home/student/JB183/labs/jaas-review/todo-roles.properties` file for authorization.
- 4.1. In the terminal window, navigate to the `/home/student/JB183/labs/jaas-review/` project directory and run the `create-sd.sh` script:

```
[student@workstation ~]$ cd JB183/labs/jaas-review
[student@workstation jaas-review]$./create-sd.sh
```

- 4.2. Confirm that the security domain is available by viewing the contents of the EAP server configuration `/opt/jboss-eap-7.0/standalone/configuration/standalone-full.xml`.
- Using a text editor, open the `/opt/jboss-eap-7.0/standalone/configuration/standalone-full.xml` configuration file and observe the new security domain that directs the application server to use the provided users and roles property files in the project directory.

```

<security-domain name="userroles" cache-type="default">
 <authentication>
 <login-module code="UsersRoles" flag="required">
 <module-option name="usersProperties" value="file:///home/student/JB183/labs/jaas-review/todo-users.properties"/>
 <module-option name="rolesProperties" value="file:///home/student/JB183/labs/jaas-review/todo-roles.properties"/>
 </login-module>
 </authentication>
</security-domain>
```

5. Update the `jboss-web.xml` file to use the `userroles` security domain.
- 5.1. Open the `jboss-web.xml` class by expanding the `jaas-review` item in the **Project Explorer** tab in the left pane of JBDS. Click on `jaas-review > Java Resources > src/main/webapp > WEB-INF` and expand it. Double-click the `jboss-web.xml` file.
- 5.2. Update the `jboss-web.xml` file to use the new security domain named `userroles`:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<jboss-web>
 <security-domain>userroles</security-domain>
</jboss-web>
```

5.3. Save your changes by pressing **Ctrl+S**.

6. Update the `web.xml` file to enable RESTEasy security annotations, use **BASIC** authentication, and restrict access to the application's REST API with the path `/api/*` to the roles manager, employee, and superuser.

- 6.1. Open the `web.xml` class by expanding the `jaas-review` item in the Project Explorer tab in the left pane of JBDS. Click on `jaas-review > Java Resources > src/main/webapp > WEB-INF` and expand it. Double-click the `web.xml` file.
- 6.2. Create a new `<context-param>` tag containing the `resteasy.role.based.security` parameter with a value of `true`:

```
<!-- Add context param -->
<context-param>
 <param-name>resteasy.role.based.security</param-name>
 <param-value>true</param-value>
</context-param>
```

- 6.3. Create a new `security-constraint` that restricts the `EmployeeRestService` using a `<url-pattern>` set to `/api/employees/*`:

```
<!-- Add security constraint -->
<security-constraint>
 <web-resource-collection>
 <web-resource-name>Secured REST Resources</web-resource-name>
 <url-pattern>/api/employees/*</url-pattern>
 </web-resource-collection>
</security-constraint>
```

- 6.4. Update the `<security-constraint>` to restrict access for users with the employee, manager, or superuser role using the `auth-constraint` tag:

```
<security-constraint>
 <web-resource-collection>
 <web-resource-name>All resources</web-resource-name>
 <url-pattern>/api/employees/*</url-pattern>
 </web-resource-collection>
 <auth-constraint>
 <role-name>employee</role-name>
 <role-name>manager</role-name>
 <role-name>superuser</role-name>
 </auth-constraint>
</security-constraint>
```

- 6.5. Define the security roles for the employee, superuser, and manager roles with a `<security-role>` tag:

```
<!-- Add security role -->
<security-role>
 <role-name>manager</role-name>
</security-role>
<security-role>
 <role-name>employee</role-name>
</security-role>
<security-role>
 <role-name>superuser</role-name>
</security-role>
```

- 6.6. Create a `<login-config>` element and set the `<auth-method>` to BASIC.

```
<!-- Add login config -->
<login-config>
 <auth-method>BASIC</auth-method>
</login-config>
```

- 6.7. Save your changes by pressing **Ctrl+S**.

7. Update the `EmployeeRestService.java` RESTEasy service class and update each method with the following requirements:

#### Method to Role Mapping

Method	Roles Allowed
<code>getEmployee(Long id)</code>	All
<code>deleteEmployee(Long id)</code>	manager, superuser
<code>saveEmployee(Employee employee)</code>	manager, superuser
<code>getEmployeesForManager(Long managerId)</code>	All
<code>createDepartment(Department d)</code>	manager, superuser
<code>createManager(Manager m)</code>	superuser
<code>assignEmployee(Long employeeId, Long departmentId)</code>	manager, superuser
<code>assignManager(Long managerId, Long departmentId)</code>	superuser

- 7.1. In the Project Explorer tab in the left pane of JBDS, select `src/main/java > com.redhat.training.rest`. Double-click `EmployeeRestService.java` to view the file.
- 7.2. Update the `getEmployee` method with a `@PermitAll` annotation to allow all roles listed in the `web.xml` to access the method:

```

@PermitAll
@GET
@Path("/byId/{id}")
@Produces(MediaType.APPLICATION_JSON)
public Employee getEmployee(@PathParam("id") Long id) {
 return employeeBean.readEmployeeById(id);
}

```

- 7.3. Update the `getEmployeesForManager` method with a `@PermitAll` annotation to allow all roles listed in the `web.xml` to access the method:

```

@PermitAll
@GET
@Path("getByManager/{managerId}")
@Produces(MediaType.APPLICATION_XML)
public List<Employee> getEmployeesForManager(@PathParam("managerId")Long
 managerId){
 Manager manager = managerBean.findById(managerId);
 return employeeBean.findAllForManager(manager);
}

```

- 7.4. Update the `assignEmployee` method with a `@RolesAllowed` annotation to allow only the `manager` and `superuser` roles listed in the `web.xml` to access the method:

```

@RolesAllowed({"manager", "superuser"})
@POST
@Path("assignEmployee/{employeeId}/{departmentId}")
public void assignEmployee(@PathParam("employeeId")Long employeeId,
 @PathParam("departmentId") Long departmentId) {
 Employee e = employeeBean.readEmployeeById(employeeId);
 Department d = departmentBean.findById(departmentId);
 e.setDepartment(d);
 employeeBean.updateEmployee(e);
}

```

- 7.5. Update the `assignManager` method with a `@RolesAllowed` annotation to allow only the `superuser` role listed in the `web.xml` to access the method:

```

@RolesAllowed({"superuser"})
@POST
@Path("assignManager/{managerId}/{departmentId}")
public void assignManager(@PathParam("managerId")Long managerId,
 @PathParam("departmentId") Long departmentId) {
 Manager m = managerBean.findById(managerId);
 Department d = departmentBean.findById(departmentId);
 m.setDepartment(d);
 managerBean.updateManager(m);
}

```

- 7.6. Save your changes by pressing `Ctrl+S`.

8. Deploy the `jaas-review` application using the following commands in the terminal window:

```
[student@workstation ~]$ cd /home/student/JB183/labs/jaas-review
[student@workstation jaas-review]$ mvn wildfly:deploy
```

9. Test the new health check REST method using the REST client Firefox plugin.
  - 9.1. Start Firefox on the **workstation** VM and click the REST Client plugin icon in the browser's toolbar.
  - 9.2. In the top toolbar, click **Headers**, and select **Custom Header** to add a new custom header to the request.
  - 9.3. In the custom header dialog, enter the following information:
    - Name: **Content-Type**
    - Value: **application/json**
  - 9.4. Select **GET** as the **Method**. In the URL form, enter `http://localhost:8080/jaas-review/api/healthCheck`. **Do not set any authentication. This service should be publicly available.**  
Click **Send**.
  - 9.5. Verify in the **Response Headers** tab that the **Status Code** is `200 OK`.
10. Use the plugin to verify that the **employee** user with password **redhat1!** has permissions to access only the **GET** methods.
  - 10.1. In the Firefox plugin, click **Authentication** and then click **Basic Authentication**. Log in with the following credentials:
    - Username: **employee**
    - Password: **redhat1!**
 Click **Okay**.
  - 10.2. Set the **Method Type** to **GET**. Set the **URL** to `http://localhost:8080/jaas-review/api/employees/getByManager/1`
  - 10.3. Click **Send**. The server returns a `200 OK` code.
  - 10.4. Repeat this step with the **byId** endpoint and make sure a `200 OK` response code is returned.
  - 10.5. Change the **Method Type** to **POST**. Set the **URL** to `http://localhost:8080/jaas-review/api/employees/assignEmployee/1/2`
  - 10.6. Click **Send**. The server returns a `403 Forbidden` code.
  - 10.7. Repeat the previous two steps with the **assignManager**, and **assignEmployee** methods and make sure each returns a `403 Forbidden` code.
11. Use the plugin to verify that the **manager** user with password **redhat1!** has permissions to access only correct methods.
  - 11.1. In the Firefox plugin, click **Authentication** and then click **Basic Authentication**. Log in with the following credentials:

- Username: **manager**
- Password: **redhat1!**

Click Okay.

- 11.2. Set the **Method Type** to **GET**. Set the URL to `http://localhost:8080/jaas-review/api/employees/getByManager/1`
  - 11.3. Click **Send**. The server returns a **200 OK** code.
  - 11.4. Repeat this step with the **byId** method and make sure a **200 OK** response code is returned.
  - 11.5. Change the **Method Type** to **POST**. Set the URL to `http://localhost:8080/jaas-review/api/employees/assignEmployee/1/2`
  - 11.6. Click **Send**. The server returns a **204 No Content** code.
  - 11.7. Repeat the previous two steps with the **assignManager** method. Make sure that each returns a **403 Forbidden** code.
12. Use the plugin to verify that the **superuser** user with password **redhat1!** has permissions to access only correct methods.
    - 12.1. In the Firefox plugin, click **Authentication** and then click **Basic Authentication**. Log in with the following credentials:
      - Username: **superuser**
      - Password: **redhat1!**
    - Click Okay.
  - 12.2. Set the **Method Type** to **GET**. Set the URL to `http://localhost:8080/jaas-review/api/employees/getByManager/1`
  - 12.3. Click **Send**. The server returns a **200 OK** code.
  - 12.4. Repeat this step with the **byId** method and make sure that a **200 OK** response code is returned.
  - 12.5. Change the **Method Type** to **POST**. Set the URL to `http://localhost:8080/jaas-review/api/employees/assignEmployee/1/2`
  - 12.6. Click **Send**. The server returns a **204 No Content** code.
  - 12.7. Repeat the previous two steps with the **assignManager** method and make sure it returns a **204 No Content** code.

## Evaluation

As the **student** user on **workstation**, run the `lab jaas-review` script with the `grade` argument to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab jaas-review grade
```

After the grading succeeds, stop the EAP server and close the project in JBDS.

This concludes the lab.

