Join the explorers, builders, and individuals who boldly offer new solutions to old problems. For open source, innovation is only possible because of the people behind it.

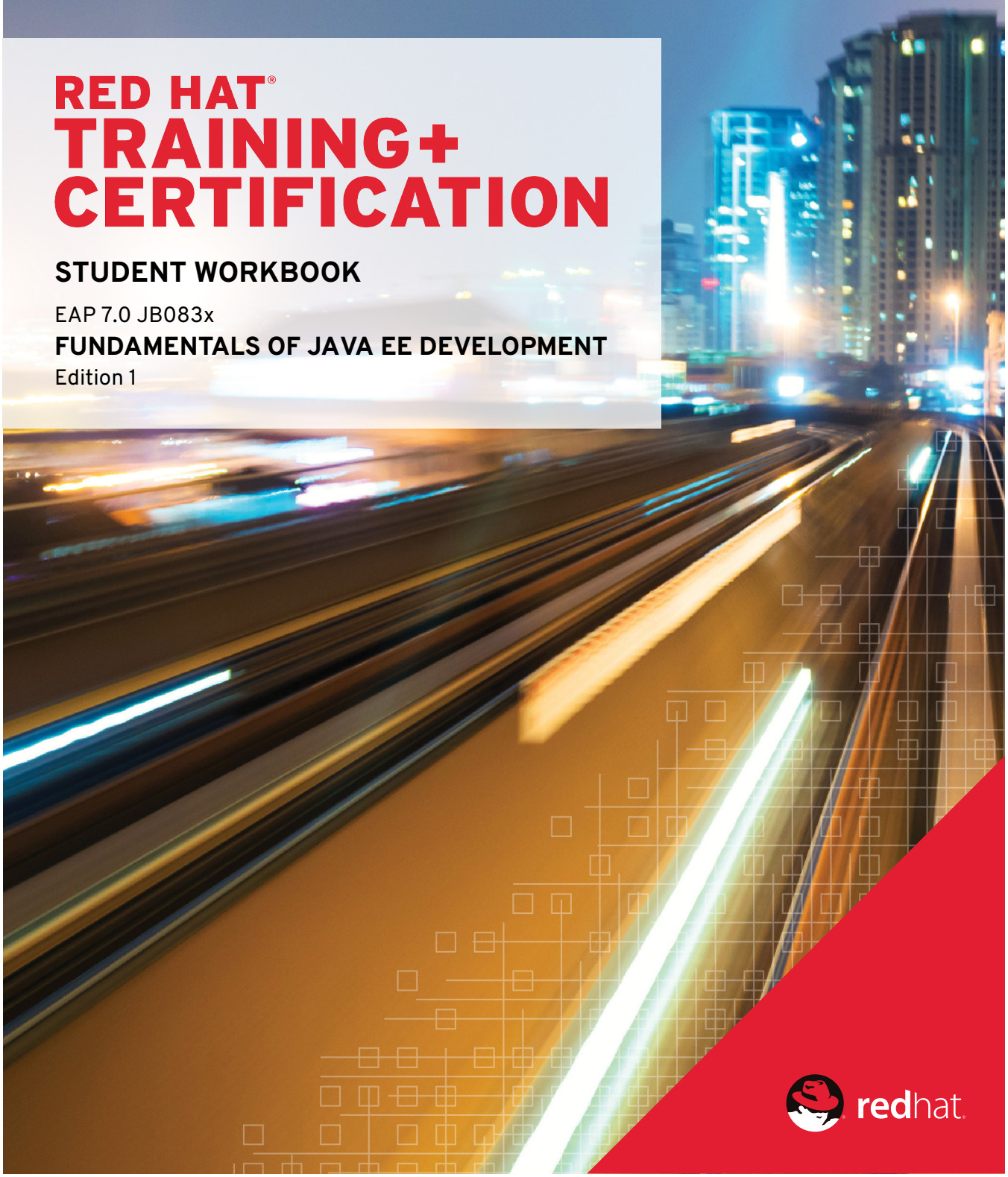
RED HAT® TRAINING+ CERTIFICATION

STUDENT WORKBOOK

EAP 7.0 JB083x

FUNDAMENTALS OF JAVA EE DEVELOPMENT

Edition 1



redhat.

JB083x-EAP-7.0-en-1-20181001

FUNDAMENTALS OF JAVA EE DEVELOPMENT

redhat.

JB083x-EAP-7.0-en-1-20181001

EAP 7.0 JB083x
Fundamentals of Java EE Development
Edition 1 20181001
Publication date 20181001

Authors: Ravishankar Srinivasan, Eduardo Ramirez, Zachary Gutterman, Jim Rigsbee, Richard Allred
Editor: David O'Brien

Contributors: Connie Petlitzer, Rob Locke, Bowe Strickland, Scott McBrien, George Hacker, Forrest Taylor

DOCUMENT CONVENTIONS

REFERENCES

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

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.

INTRODUCTION

FUNDAMENTALS OF JAVA EE DEVELOPMENT

Fundamentals of Java EE Development (JB083x) 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 learn how to develop multi-tiered enterprise applications using various Java EE APIs.

COURSE OBJECTIVES

- Describe the differences between Java SE and Java EE application architectures.
- Create application components using the EJB, JPA, JAX-RS, and CDI specifications.
- Develop back-end components necessary to support a three-tiered web application using JBoss Enterprise Application Platform (EAP) and Apache 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.

LAB SETUP INSTRUCTIONS FOR EXERCISES

Introduction

This course includes a number of guided exercises, which give you an opportunity to practice the skills you are learning in the course presentations. To complete these exercises, you need to configure a practice environment that you completely control.

These instructions assume that you are running a recent version of Apple macOS, Microsoft Windows 7, 8, or 10, or a Linux distribution on your workstation. If you are using a Linux distribution, we recommend that you use the latest stable Red Hat Enterprise Linux (RHEL), or Fedora Workstation edition.

Hardware Requirements

The following minimum hardware is required to run the exercises in this course:

- 64-bit quad core CPU
- 4 GB RAM (minimum), 8 GB RAM (recommended)
- 80GB hard disk space

Installation Overview

You will use the following software components during this course:

- Git client
- JDK 1.8
- Apache Maven 3.3.9
- JBoss EAP 7.0.0
- Red Hat JBoss Developer Studio 11.0.0
- Firefox web browser 62.0 or later with the REST client add-on

The installation of the lab environment consists of the following tasks:

1. Install a Git client
2. Install JDK 1.8
3. Install and configure Apache Maven
4. Install Red Hat JBoss EAP
5. Install and configure Red Hat JBoss Developer Studio
6. Install and configure Firefox 62.0 or later with the REST client add-on

These instructions were tested on the following configurations:

- Apple macOS 10.12, Oracle JDK 1.8, Git 2.10
- Microsoft Windows 10 Pro, Oracle JDK 1.8, Git 2.18

- Red Hat Enterprise Linux 7, OpenJDK 1.8, Git 1.8
- Fedora 28, OpenJDK 1.8, Git 2.17

For all the above listed operating systems, JBoss EAP 7.0.0, Apache Maven 3.3.9, Firefox 62.0, and Red Hat JBoss Developer Studio 11.0 were used.

You can download a free copy of Red Hat Enterprise Linux, Red Hat JBoss EAP, and Red Hat JBoss Developer Studio IDE for development purposes from the Red Hat Developer Program [https://developer.redhat.com] portal.

INSTALLING A GIT CLIENT

Linux distributions usually include a Git client by default. If Git is not installed, install it using your Linux distribution's package manager. For example, on Fedora systems, you can install Git using the **dnf** command:

```
# dnf install git
```

For macOS and Microsoft Windows, install the Git client from the Git website. Download the client from https://git-scm.com/downloads.

INSTALLING A JDK

The first step in the lab environment set up process is to install a JDK for your operating system.

Install JDK 1.8 on Linux

For Linux distributions, OpenJDK is the best choice for a JDK 1.8 compatible Java development environment.

If you are using Red Hat Enterprise Linux, run the following command as the **root** user to install OpenJDK 1.8:

```
# yum install java-1.8.0-openjdk-devel
```

If you are using Fedora Workstation, run the following command to install OpenJDK 1.8:

```
# dnf install java-1.8.0-openjdk-devel
```

For Ubuntu and Debian-based systems, run the following command:

```
$ sudo apt install openjdk-8-jdk
```

For all other Linux distributions, consult your distribution's documentation or man pages and install OpenJDK using your distribution's package manager. OpenJDK is packaged in almost all the popular Linux distributions.

After you have installed the JDK, set the **JAVA_HOME** environment variable to point to the path where the JDK is installed. Edit the **$HOME/.bashrc** file and add the full path to the JDK installation. On RHEL and Fedora-based systems, OpenJDK is installed in the **/usr/lib/jvm/filename** folder. Add the following to the **$HOME/.bashrc** file:

```
export JAVA_HOME="/usr/lib/jvm/java-1.8.0-openjdk-<version>"
```

Replace *<version>* with the version of OpenJDK that is installed on your system.

For other Linux distributions, consult the documentation, or use the package manager tools to identify where OpenJDK is installed, and set the **JAVA_HOME** environment variable appropriately.

Install JDK 1.8 on Microsoft Windows 7/10

Download Oracle JDK 1.8 64-bit installer for Windows from the Oracle website at:

https://www.oracle.com/technetwork/java/javase/downloads/index.html

The installer executable file will be in the format **jdk-8u<*version*>-windows-x64.exe**. Run the downloaded installer to install the JDK. Ensure that the directory name where the JDK is installed does not have spaces in it.

Add a new environment variable called **JAVA_HOME**, and set the value to the directory where you installed the JDK. Append the **JAVA_HOME/bin** directory path to the **PATH** environment variable to ensure that you can run the **javac** and **java** commands from the command line.

Install JDK 1.8 on Apple macOS

Download the Oracle JDK 1.8 64-bit installer for macOS from the Oracle website at:

https://www.oracle.com/technetwork/java/javase/downloads/index.html

The installer will be in the format **jdk-8u<*version*>-macosx-x64.dmg**. Run the installer and install the JDK. Use the default values during the installation.

After you have installed the JDK, set the **JAVA_HOME** environment variable to point to the path where the JDK is installed. Edit the **$HOME/.bash_profile** file and add the full path to the JDK installation. On macOS, the JDK is installed in the **/usr/libexec** directory. Add the following to the **$HOME/.bash_profile** file:

```
export JAVA_HOME="$(/usr/libexec/java_home -v 1.8)"
```

Append the **JAVA_HOME/bin** directory path to the **PATH** environment variable to ensure that you can run the **javac** and **java** commands from the command line:

```
export PATH="$JAVA_HOME/bin:$PATH"
```

INSTALLING AND CONFIGURING APACHE MAVEN

You will use Apache Maven throughout this course to build, package, test, and deploy applications on JBoss EAP. The installation process remains the same for all operating systems: download the Apache Maven 3.3.9 binary distribution from https://repo.maven.apache.org/maven2/org/apache/maven/apache-maven/3.3.9/, and extract to a directory of your choice. Make sure you do not have spaces in the directory name.

After you have installed Maven, set the **M2_HOME** environment variable to point to the path where Maven is installed.

Append the **M2_HOME/bin** directory path to the **PATH** environment variable to ensure that you can run the **mvn** command from the command line.

To verify that Maven is installed correctly, run the **mvn -v** command. For example, on a Fedora 28 system, the following is the output:

```
$ mvn -v
Apache Maven 3.3.9...
Maven home: /home/user1/apps/apache-maven-3.3.9
```

```
Java version: 1.8.0_181, vendor: Oracle Corporation
Java home: /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.181-7.b13.fc28.x86_64/jre
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "4.17.19-200.fc28.x86_64"...
```

Configuring Maven Repositories

After Maven is installed, configure the repositories from where Maven should fetch dependencies for building your applications.

Download the **`settings.xml`** file from https://raw.githubusercontent.com/RedHatTraining/JB083x-lab/master/settings.xml.

This file contains the location of the repositories from where Maven should fetch dependencies. Copy it to your local system as follows:

- For Microsoft Windows 7 and 10 systems, create a new directory called **`.m2`** in your home directory and copy the **`settings.xml`** file to this new directory. For example, if the user name on your local Windows system is **`user1`**, the **`settings.xml`** file should be at **`C:\Users\user1\.m2\settings.xml`**.

- For Linux-based systems, create a new directory called **`.m2`** in your home directory and copy the **`settings.xml`** to this new directory. For example, if the user name on your local Linux system is **`user1`**, the **`settings.xml`** file should be at **`/home/user1/.m2/settings.xml`**.

- For macOS systems, create a new directory called **`.m2`** in your home directory and copy the **`settings.xml`** to this new directory. For example, if the user name on your local macOS system is **`user1`**, the **`settings.xml`** file should be at **`/Users/user1/.m2/settings.xml`**.

INSTALLING RED HAT JBOSS EAP

Register for a free developer account at https://developers.redhat.com.

Download the EAP 7.0.0 installer JAR file from https://developers.redhat.com/download-manager/file/jboss-eap-7.0.0-installer.jar.

To install JBoss EAP, follow the instructions from the JBoss EAP installation guide at:

https://access.redhat.com/documentation/en-us/red_hat_jboss_enterprise_application_platform/7.0/html-single/installation_guide/#installer_installation

INSTALLING RED HAT JBOSS DEVELOPER STUDIO

Download the Red Hat Developer Studio 11.0.0 installer JAR file from https://developers.redhat.com/download-manager/file/devstudio-11.0.0.GA-installer-standalone.jar.

To install the IDE, follow the instructions from the Red Hat Developer Studio installation guide at:

https://access.redhat.com/documentation/en-us/red_hat_jboss_developer_studio/11.3/html-single/installation_guide/#installer

INSTALLING A WEB BROWSER REST CLIENT

The chapter on developing and testing RESTful services requires a REST client add-on for a web browser to simplify testing. We recommend you install the REST client add-on for Firefox.

Install Firefox 62.0 or later either by using your package manager on Linux distributions, or by downloading the installer from the Mozilla website at `www.mozilla.org`. After Firefox has been installed, download the REST client add-on and install it from the following URL:

https://addons.mozilla.org/en-US/firefox/addon/restclient

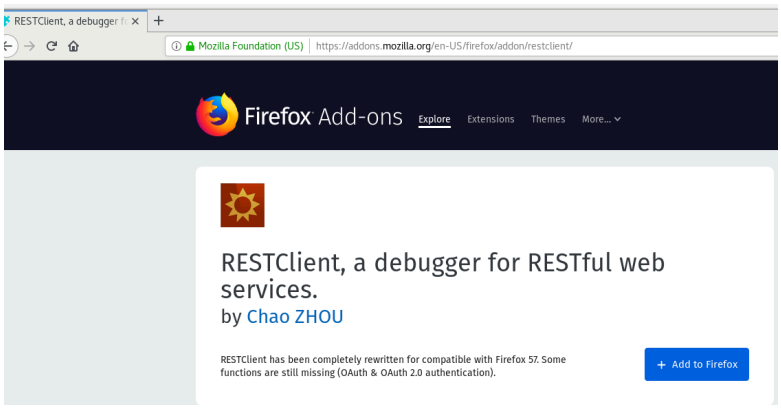


Figure 0.1: Firefox REST client add-on

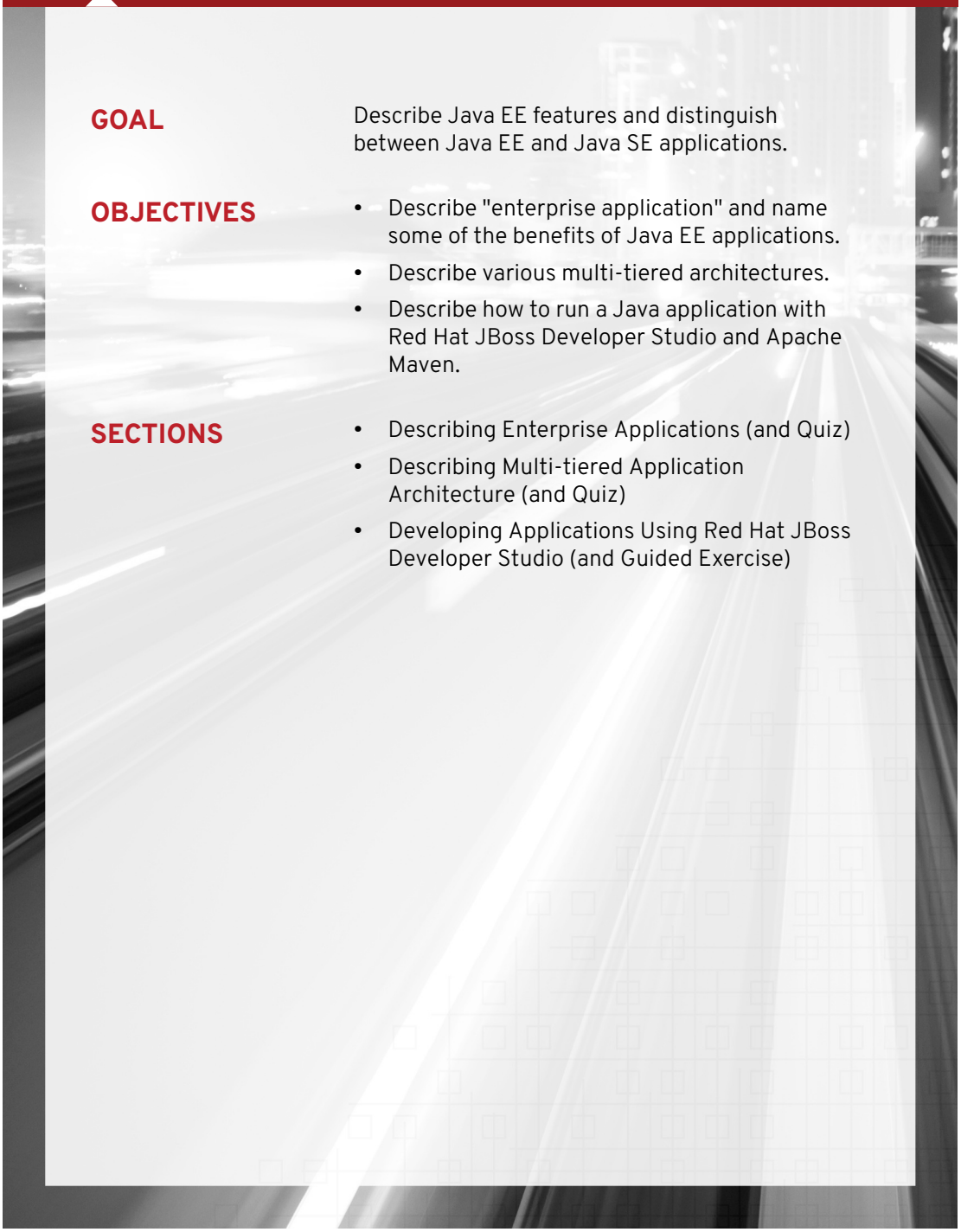
Click **Add to Firefox** to install the add-on.

If you do not want to use Firefox, use the links below to download alternative REST clients:

- For macOS, download Cocoa REST client App [https://mmattozzi.github.io/cocoa-rest-client/], or The Postman App [https://www.getpostman.com/].
- For Microsoft Windows systems, download Postman [https://www.getpostman.com/], or Insomnia [https://insomnia.rest/].
- For Google Chrome, download Advanced REST client [https://install.advancedrestclient.com], or Postman [https://chrome.google.com/webstore/detail/postman/fhbjgbiflinjbdggehcddcbncdddomop].

CHAPTER 1

TRANSITIONING TO MULTI-TIERED APPLICATIONS



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.
- Describe various multi-tiered architectures.
- Describe how to run a Java application with Red Hat JBoss Developer Studio and Apache Maven.

SECTIONS

- Describing Enterprise Applications (and Quiz)
- Describing Multi-tiered Application Architecture (and Quiz)
- Developing Applications Using Red Hat JBoss Developer Studio (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.

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.

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* 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 Red Hat JBoss Developer Studio also have native support for Maven built-in by default. All Maven tasks can be run from within the IDE itself without using the command line.

REFERENCES

JCP
https://www.jcp.org/en/home/index

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-us/red_hat_jboss_enterprise_application_platform/7.0/html-single/7.0.0_release_notes

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.

5. 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.

6. 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 invoked from the command line.

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.

5. 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.

6. 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 invoked from the command line.

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.

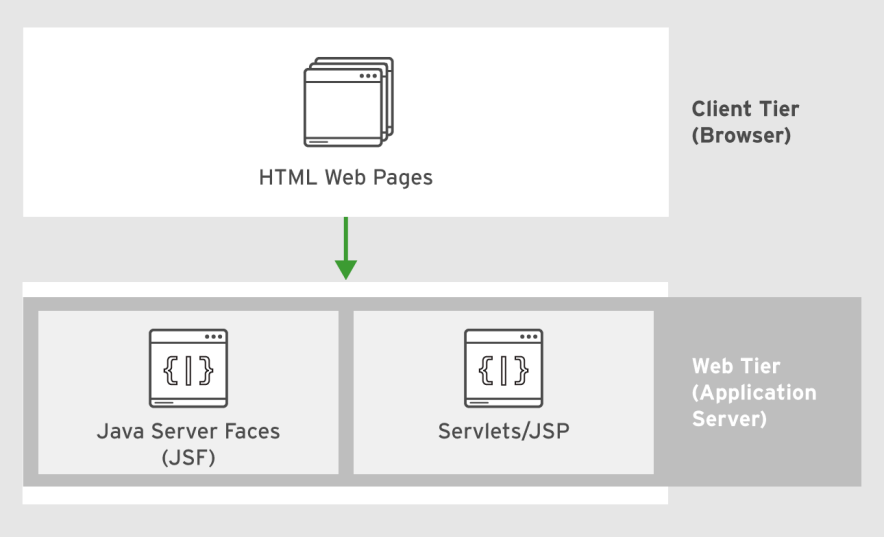


Figure 1.1: Simple Web-Centric architecture

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.

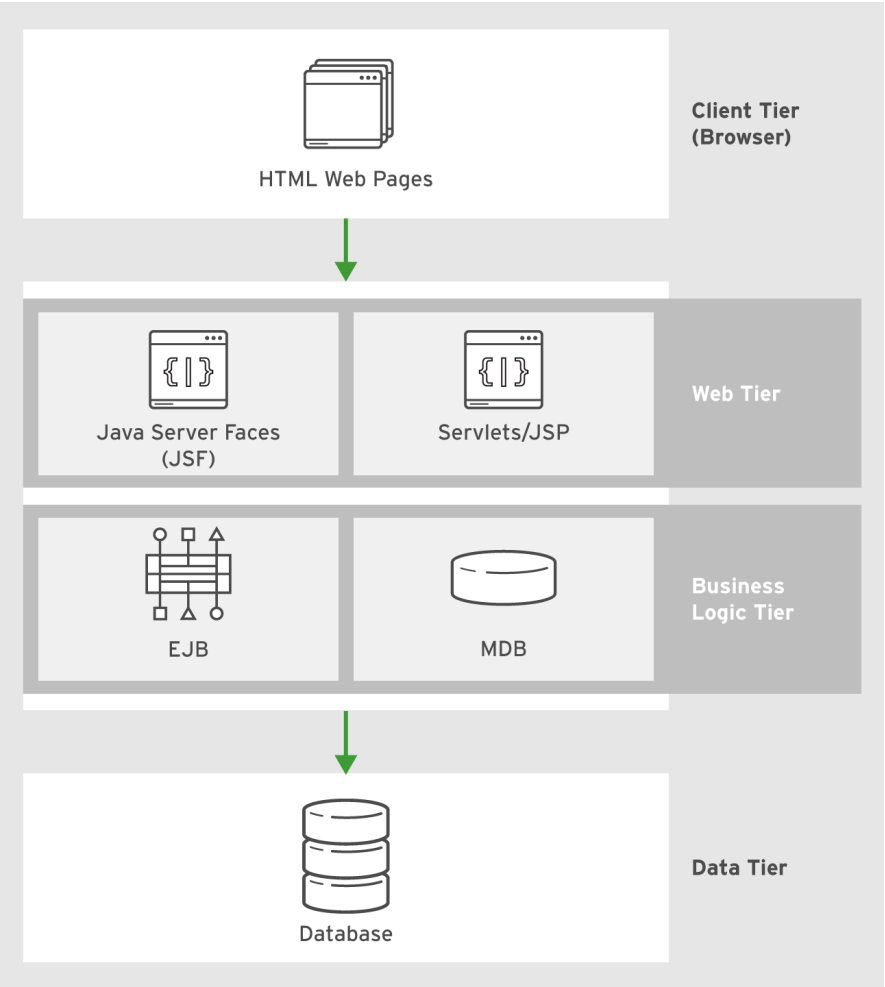


Figure 1.2: Combined web and business logic component-based architecture

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).

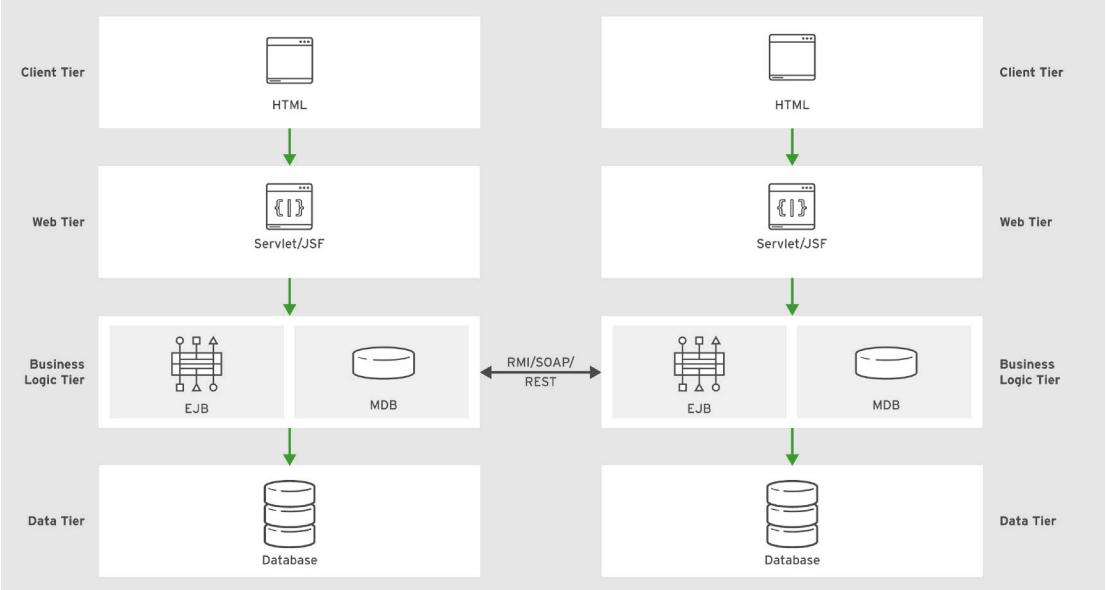


Figure 1.3: Business-to-Business architecture

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.

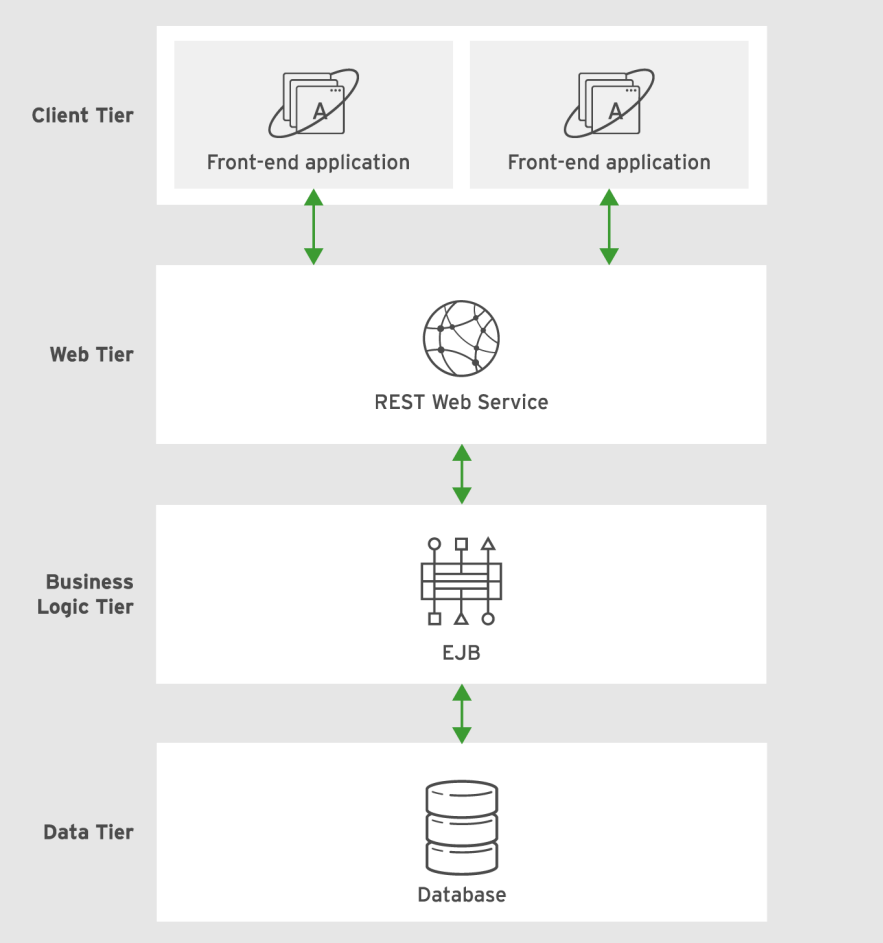


Figure 1.4: Simple web service application architecture

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.

DEVELOPING APPLICATIONS USING RED HAT JBOSS DEVELOPER STUDIO

OBJECTIVES

After completing this section, students should be able to:

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

RED HAT JBOSS DEVELOPER STUDIO

Red Hat JBoss Developer Studio 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. It 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.

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	**`src/main/java`**	The directory contains Java classes included in **`WEB-INF/classes`** for a WAR or root of a JAR.
Configuration Files	**`src/main/resources`**	The directory contains configuration files included in **`WEB-INF/classes`** for a WAR or root of JAR.
Java Test Code	**`src/test/java`**	The directory contains the test source code.
Test Configuration Files	**`src/test/resources`**	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 **`import`** 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 the Java EE container provides the dependency at runtime. It is used during build and test.

Maven is integrated in Red Hat JBoss Developer Studio, 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 wildfly: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 Red Hat 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 developing a number of applications that will be deployed on a JBoss EAP 7 application server. These applications will use several APIs from the Java EE 7 specification. You will make use of Maven and Red Hat JBoss Developer Studio extensively in this course to manage application packaging and deployment.

To build, package, and run a standalone application that uses only the Java SE API with Maven, you will run the following commands:

```
$ mvn clean package
$ java -jar target/myapp.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 applications are built and deployed to JBoss EAP by using the following command:

```
$ 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 JBoss 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

GUIDED EXERCISE

DEVELOPING APPLICATIONS USING RED HAT JBOSS DEVELOPER STUDIO

In this exercise, you will build and run a simple command-line application using Red Hat JBoss Developer Studio and Apache Maven.

OUTCOMES

You should be able to import the source code of a simple command-line application into Red Hat JBoss Developer Studio and run it using Maven.

BEFORE YOU BEGIN

The source code for the command-line application is available in a Git repository.

Open a terminal window on your system, and run the following command to download the lab files required for this workshop.

```
$ git clone https://github.com/RedHatTraining/JB083x-lab
```

The above command creates a directory called **`JB083x-lab`**. This directory contains the source code for all the applications used in this course. There are two subdirectories in this directory named **`labs`** and **`solutions`**, which contain the source code for all the labs, and the corresponding solution files for the labs in this course.

IMPORTANT

You only need to clone the Git repository once during the entire course. You do not have to clone the repository for every exercise in this course.

The source code for the application used in this exercise is in the **`labs/todojse`** directory.

1. Import the **todojse** project into Red Hat JBoss Developer Studio.
   1.1. Start the Red Hat JBoss Developer Studio IDE.
   1.2. Select a workspace folder.

   In the **Eclipse Launcher** window, accept the default value in the **Workspace** field, select **Use this as the default and do not ask again**, and then click **Launch**.

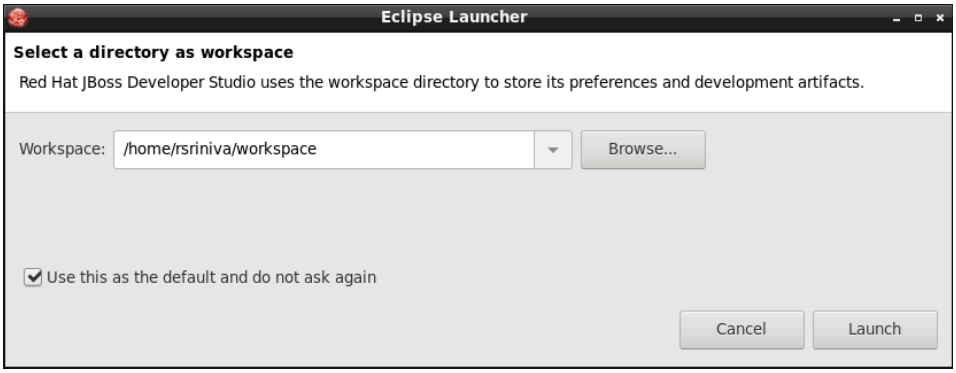


Figure 1.5: Select workspace

NOTE

The default workspace path may be different from the one shown in the screenshot depending on your operating system and the user account with which you are running this exercise.

1.3. In the IDE 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 **JB083x-lab** → **labs** → **todojse** directory, and then click **OK**.

On the **Maven projects** page, click **Finish**.

1.5. Watch the IDE 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 displays 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 project dependencies (libraries, frameworks, and modules that this project depends on). In this case, no dependencies on any external libraries exist; it only utilizes 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 is **`jar`**. Maven ensures that when the project is built it creates 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 you can 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>
```

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 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. These methods implement how tasks are created, listed, marked as completed, deleted, and found respectively.

These methods are invoked from the main runnable class, depending on which menu option the user selects. If they select 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. Use Maven to build and run the todojse application from the command line.

4.1. Before you build and run the application from within the IDE, use Maven to build and run it from the command line.

Open a new terminal window and navigate to the **`JB083x-lab/labs/todojse`** folder. You can now build and package the application as a JAR file using Maven's **`package`** goal.

Use the following command to build the application:

```
[student@localhost todojse]$ mvn clean package
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building todojse 1.0
[INFO] ------------------------------------------------------------------------
[INFO]
...

[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ todojse ---
[INFO] Building jar: ...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 **JB083x-lab/labs/todojse/target** folder.

4.2. Run the application using the Maven **exec** plug-in:

```
[student@localhost 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:
```

4.3. Explore the application functionality by creating, completing, reading, listing, and deleting a few to do items. Press 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@localhost todojse]$ java -jar target/todojse-1.0.jar
```

5.2. Verify that the application launches and that the main menu is displayed.

6. Build and run the **todojse** application from within the IDE.

6.1. You can use the built-in Maven plug-in in the IDE to build, package, and run the application.

The Maven plug-in in the IDE ships 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 the **todojse** project in the **Project Explorer** and then 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:

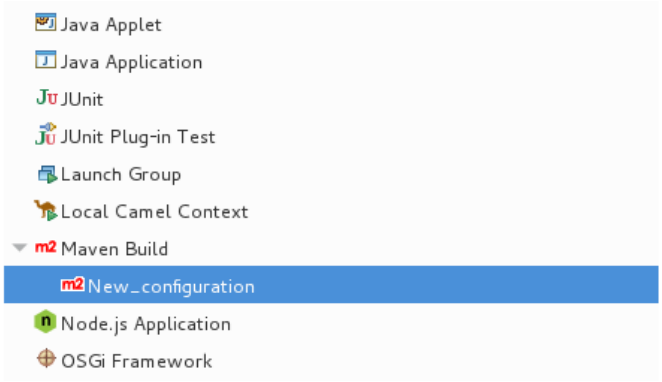


Figure 1.6: Maven run configuration

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

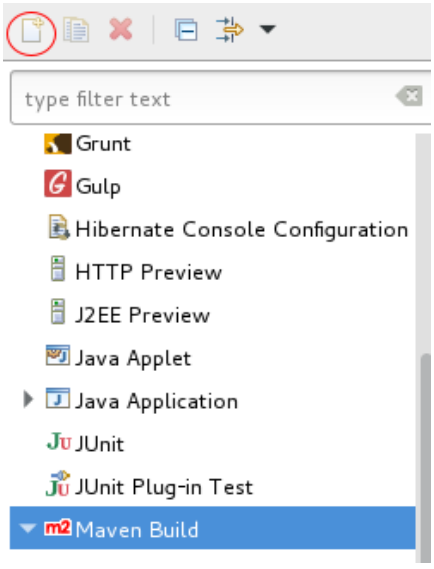


Figure 1.7: New run configuration

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

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

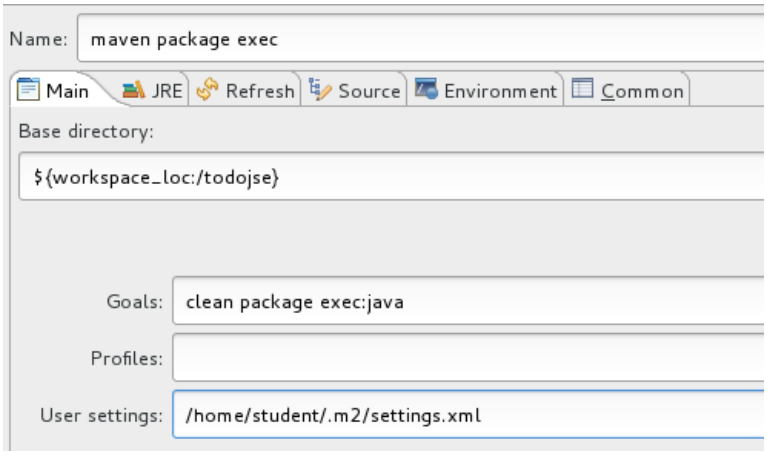


Figure 1.8: Run configuration details for Maven

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

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

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

6.6. Press **Q** in the To Do List Application main menu to exit the application.

6.7. Right-click the **todojse** project in the **Project Explorer**, and select **Close Project** to close this project.

This concludes the guided exercise.

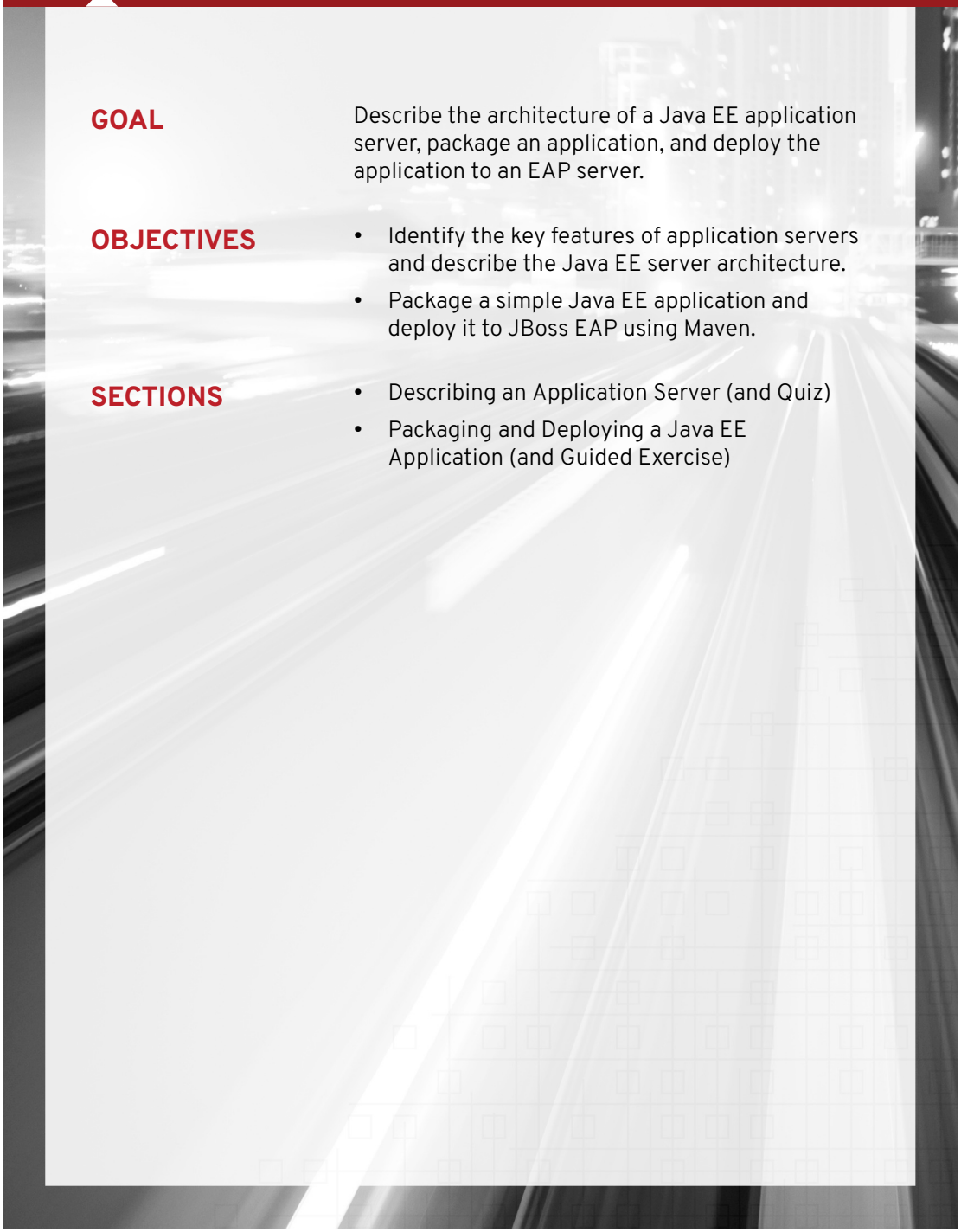
SUMMARY

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 APPLICATIONS TO AN APPLICATION SERVER



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.
- Package a simple Java EE application and deploy it to JBoss EAP using Maven.

SECTIONS

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

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.

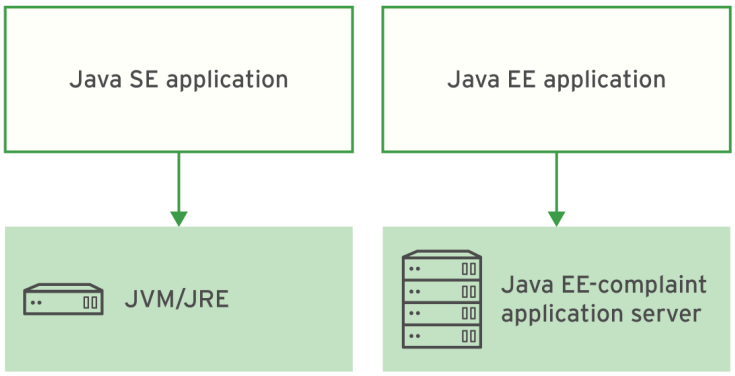


Figure 2.1: Java SE versus Java EE applications

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.
- 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`** subsytem.

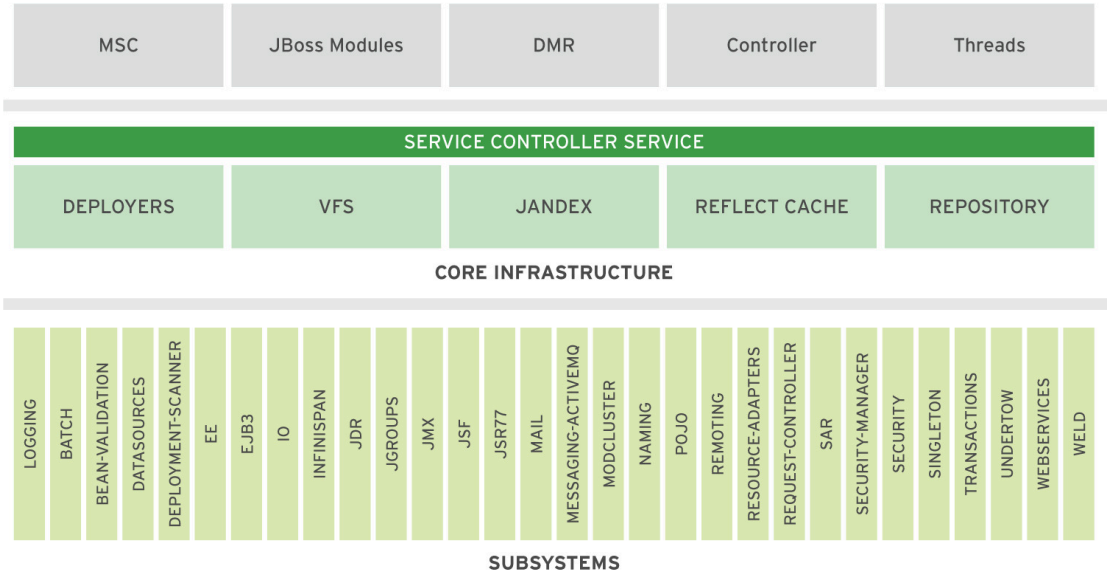


Figure 2.2: EAP 7 architecture

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 **`Classloader`**, 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.

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.

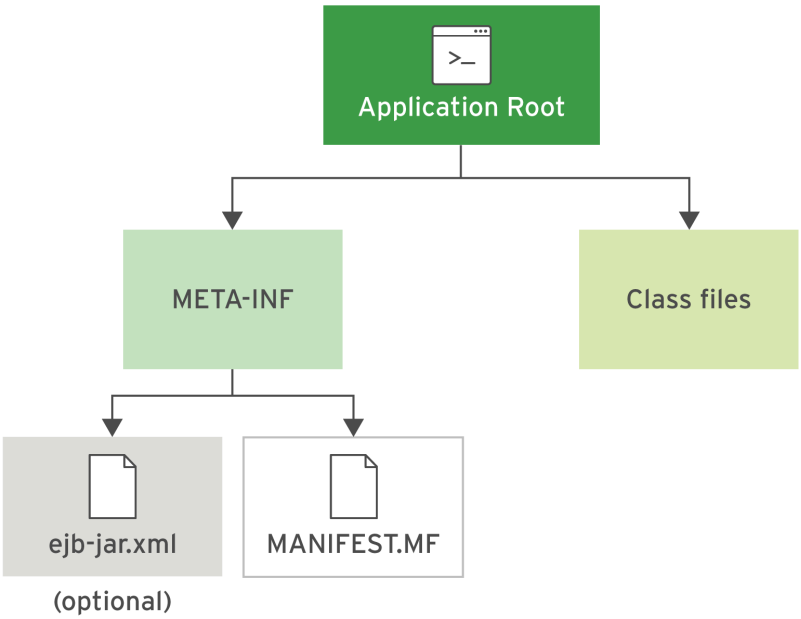


Figure 2.3: Sample EJB JAR file structure

- **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.

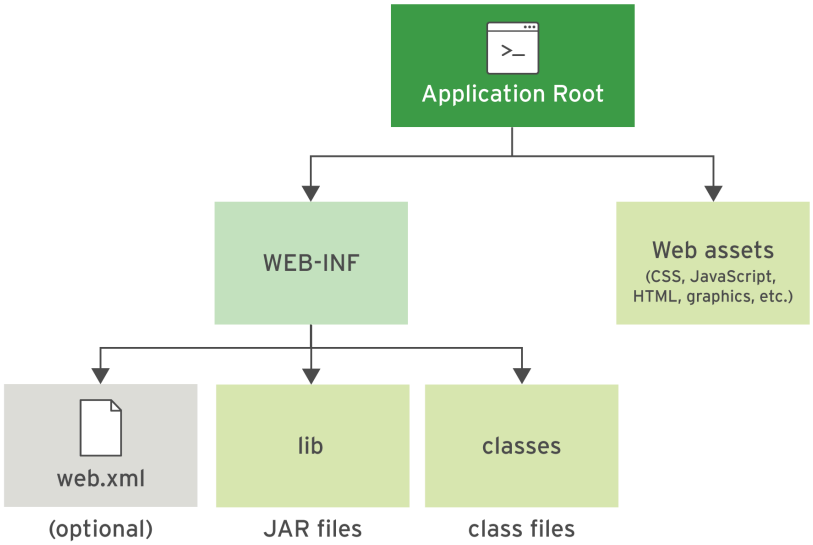


Figure 2.4: Sample WAR file structure

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

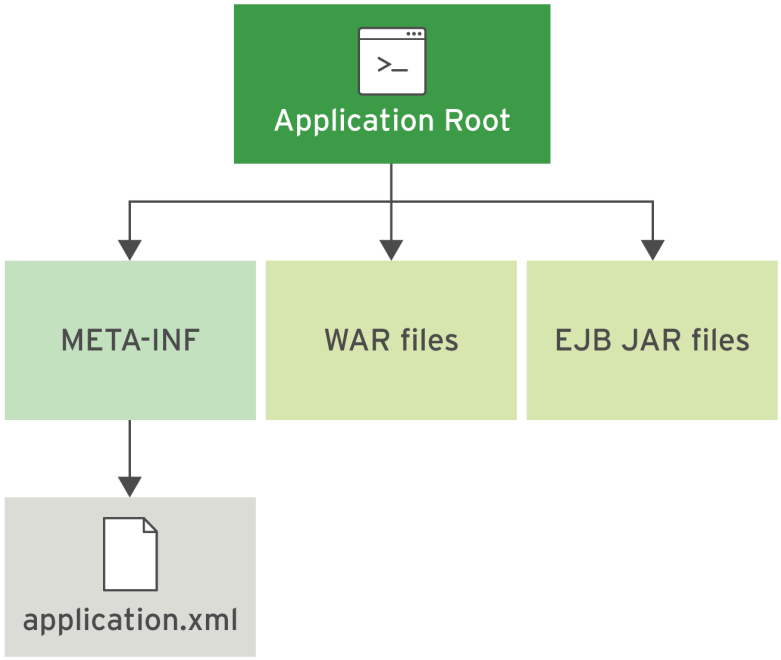


Figure 2.5: Sample EAR file structure

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:

```
$ mvn clean package wildfly:deploy
```

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

```
$ mvn clean 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/

GUIDED EXERCISE

PACKAGING AND DEPLOYING A JAVA EE APPLICATION

In this exercise, you will package and deploy a Java EE application to EAP.

OUTCOMES

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

BEFORE YOU BEGIN

The source code for the web application is available in a Git repository.

If you have not done so already, open a terminal window on your system and run the following command to download the lab files required for this course.

```
$ git clone https://github.com/RedHatTraining/JB083x-lab
```

The above command creates a directory called **`JB083x-lab`**. This directory contains the source code for all the applications used in this course. There are two subdirectories in this directory named **`labs`** and **`solutions`**, which contain the source code for all the labs, and the corresponding solution files for the labs in this course.

The source code for the web application used in this exercise is in the **`labs/hello-web`** directory.

1. Import the **`hello-web`** project into Red Hat JBoss Developer Studio IDE.
   1.1. Launch the Red Hat JBoss Developer Studio IDE.
   1.2. In the IDE 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 **`JB083x-lab/labs/hello-web`** directory, and then click OK.
   1.5. On the Maven projects page, click Finish.
   1.6. Watch the IDE 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 project dependencies (the libraries, frameworks, and modules that this project depends on).

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>
```

(1) The **`packaging`** format is declared as **`war`**. Maven ensures that when the project is built it creates 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 in the **`JB083x-lab/labs`** folder. The parent POM file declares many attributes and properties that can be used by multiple child projects that reference 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 **`JB083x-lab/labs/pom.xml`** parent POM file using the IDE (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 **`7.0.2.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 the running EAP server instance.

3. Configure an EAP server instance in the IDE.

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

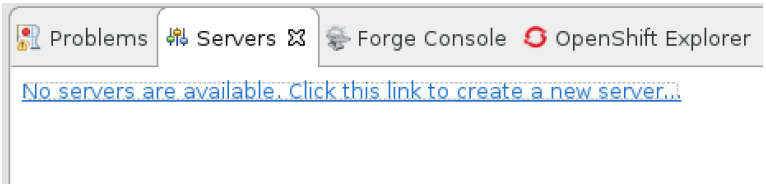


Figure 2.6: IDE servers tab

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**.

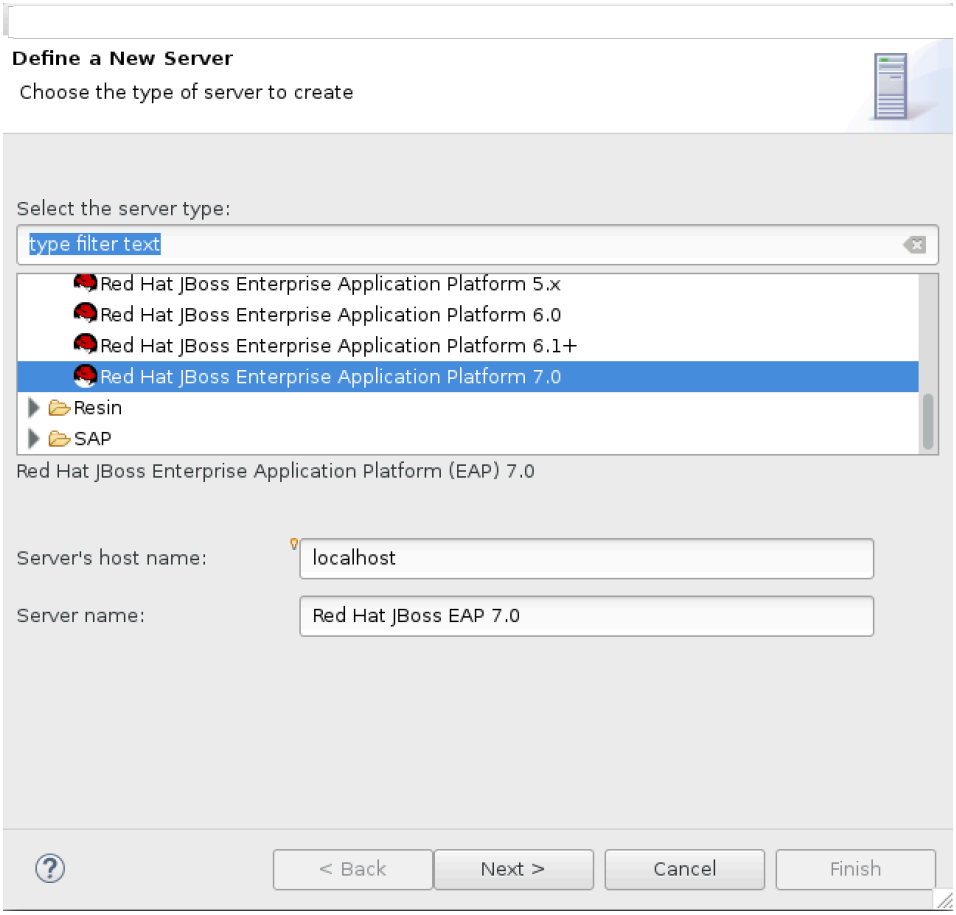


Figure 2.7: Defining a new EAP server

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**.

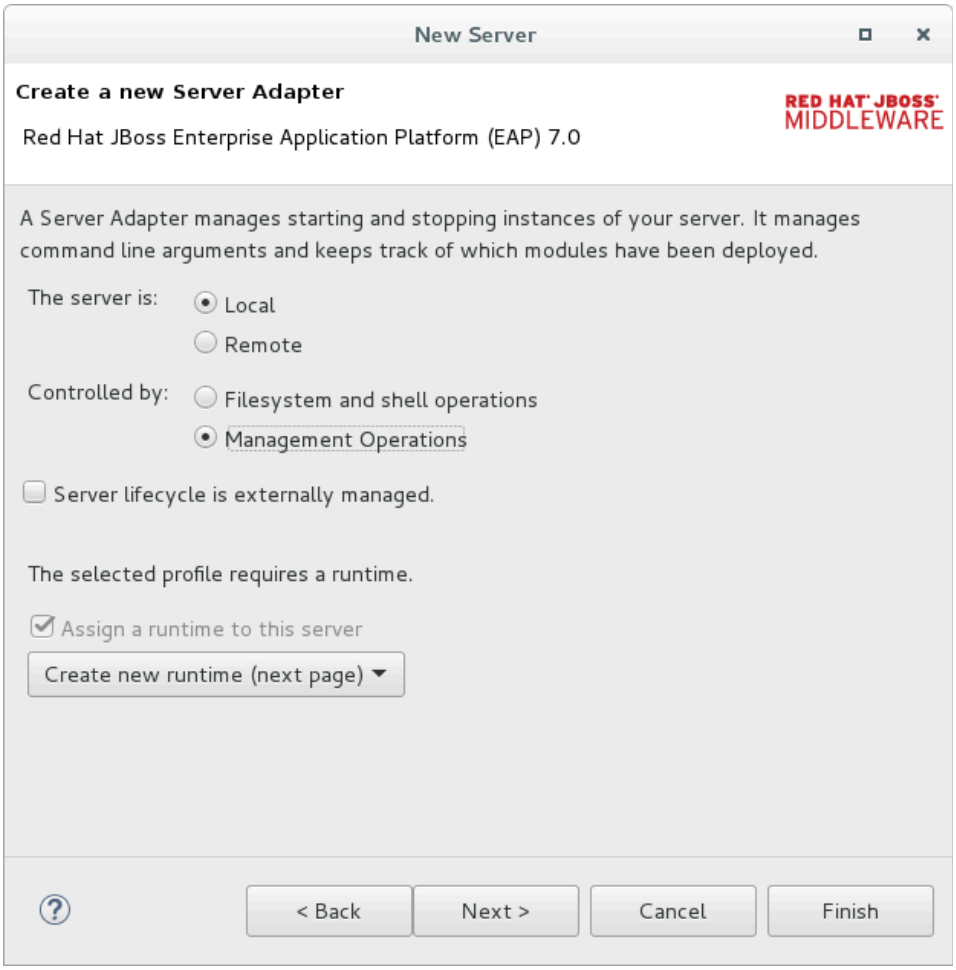


Figure 2.8: New server adapter configuration

3.5. In the JBoss Runtime window, click Browse next to the Home Directory field and navigate to the folder where you installed JBoss EAP during lab environment set up.

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.

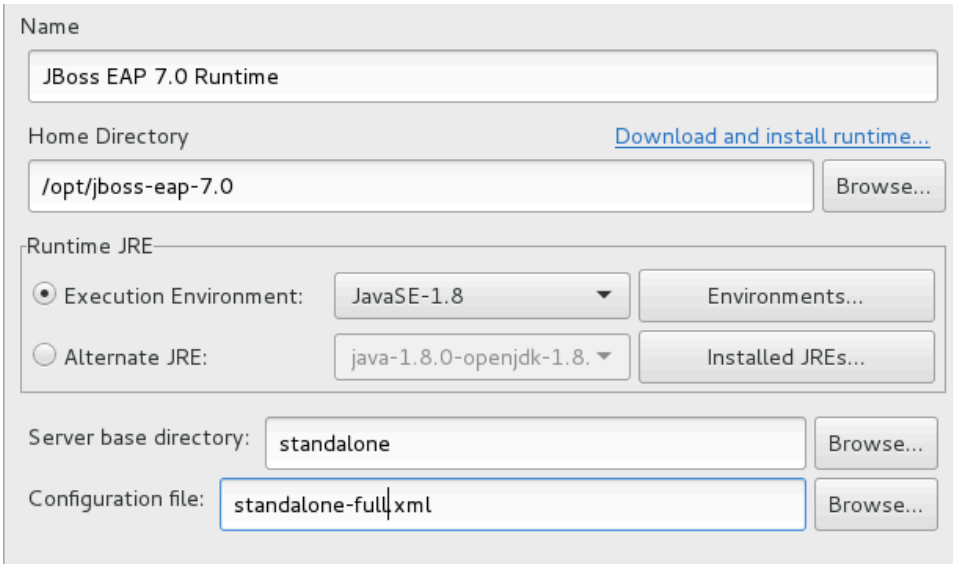


Figure 2.9: JBoss EAP runtime configuration

3.6. In the **Add and Remove** window, 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 the IDE. Click this entry to expand it.

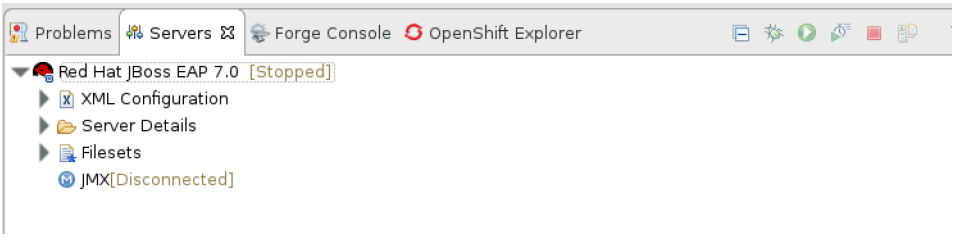


Figure 2.10: Adding a new JBoss EAP server

4. Start EAP from within the IDE.

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.

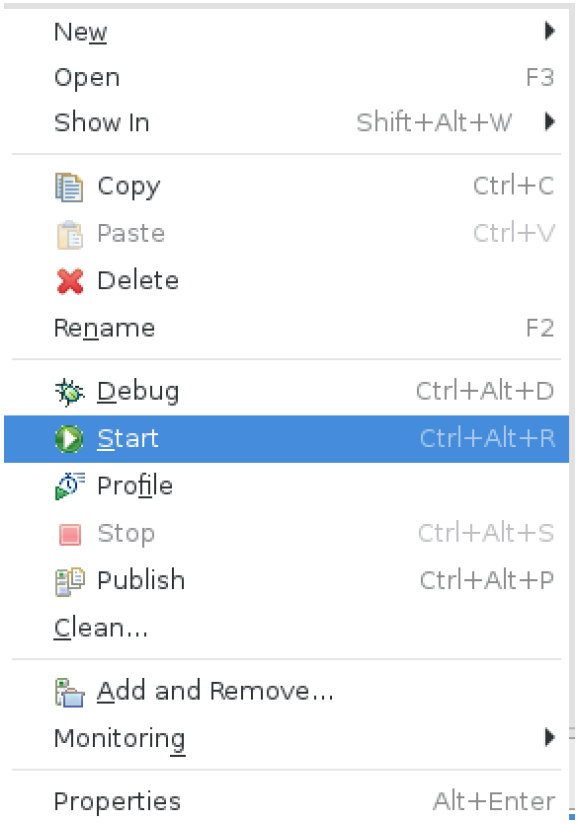


Figure 2.11: Starting a JBoss EAP server

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

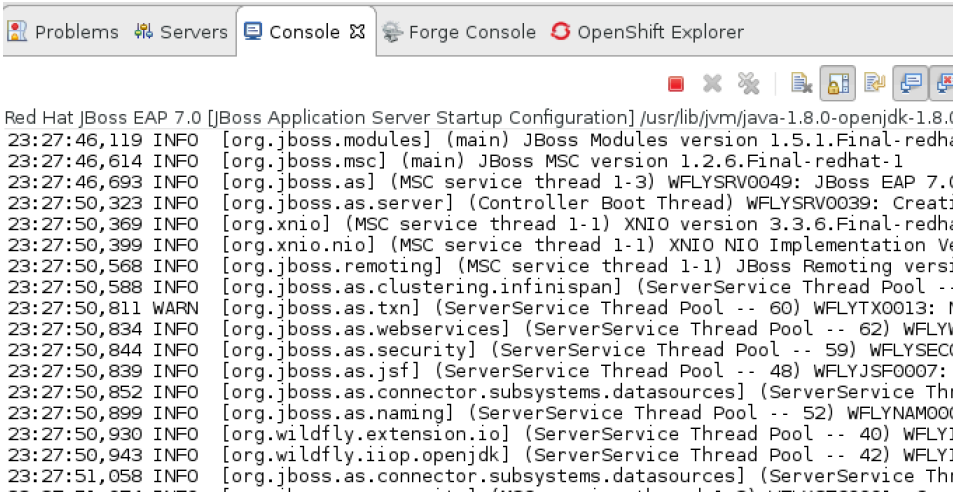


Figure 2.12: JBoss EAP console log output

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

There are two ways in which you can deploy the web application:

Option 1: Using Maven on the command line.

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

```
$ cd JB083x-lab/labs/hello-web
$ mvn clean package wildfly:deploy
```

When you run the above command, you see output similar to 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 the IDE, 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")
...output omitted...
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")
```

5.3. Access the **hello-web** application using a browser.

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

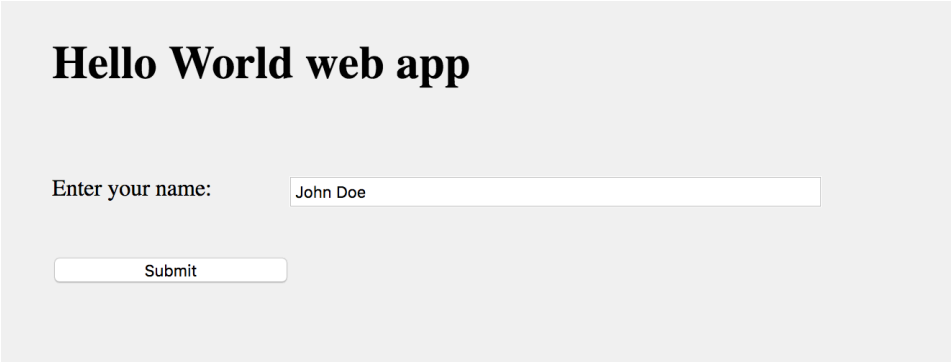


Figure 2.13: The hello-web application

5.4. Enter **John Doe** in the Enter your name field and click Submit.

5.5. Verify that the server processes the input and responds with a Hello message, as well as the current time on the server.

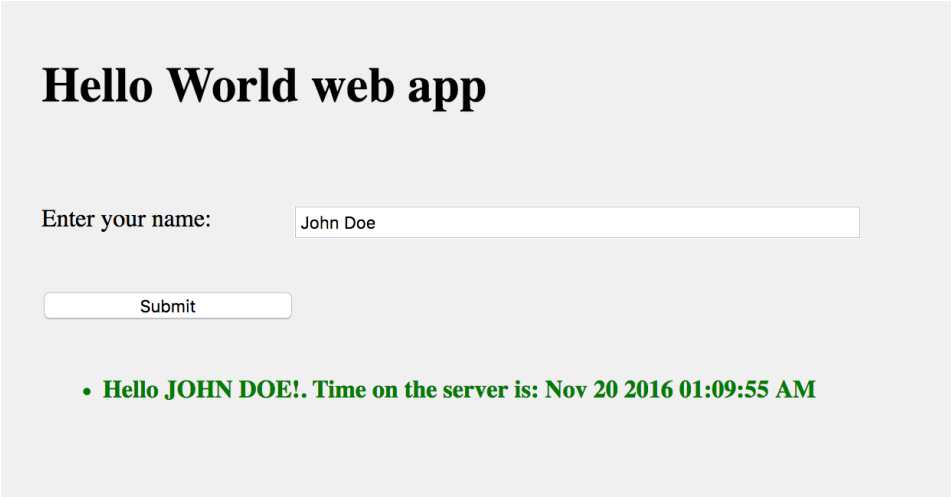


Figure 2.14: The hello-web application response

5.6. Undeploy the application and stop EAP.

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

```
$ mvn clean 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")
```

6. You can also deploy the web application from within the IDE, without using the command line.

Option 2: Using Red Hat JBoss Developer Studio

6.1. Right-click the **`hello-web`** project and select Run As → Run on Server.

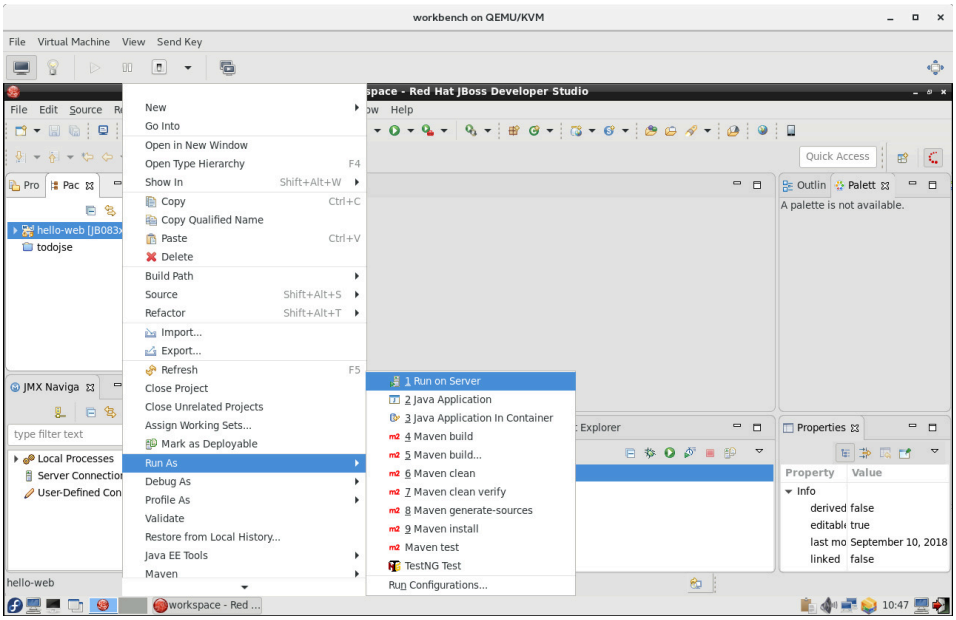


Figure 2.15: Deploying applications from within the IDE

6.2. In the **Run on Server** window, ensure that the JBoss EAP 7.0 instance is selected. Click **Next**.

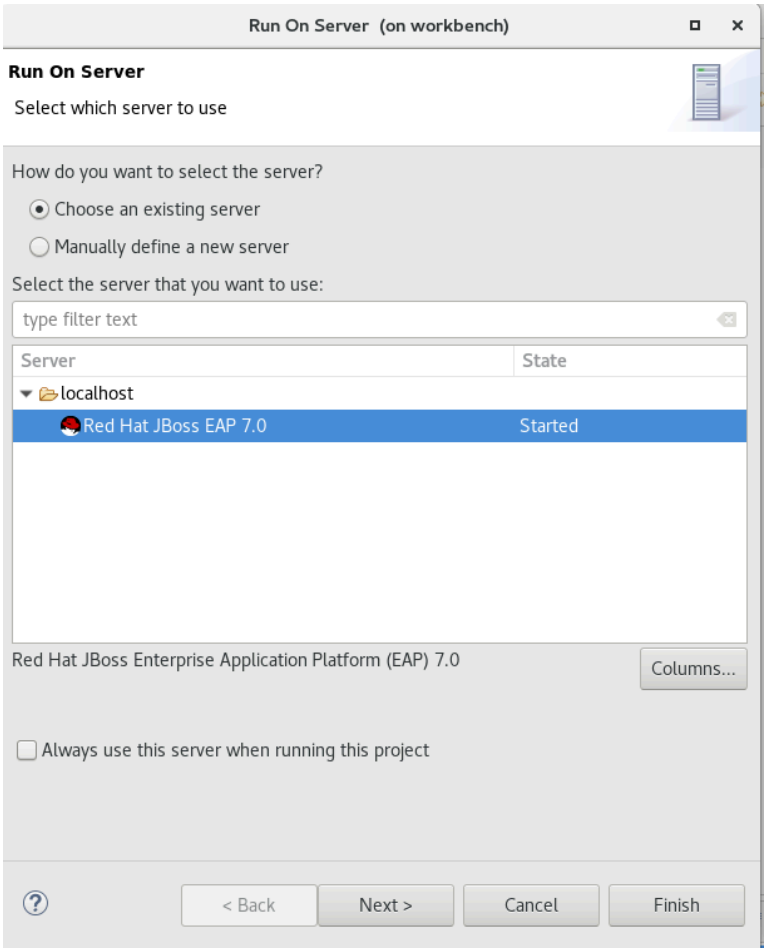


Figure 2.16: Run on Server window

6.3. In the Add and Remove window, ensure that the **`hello-web`** application is selected in the Configured column, and then click Finish.

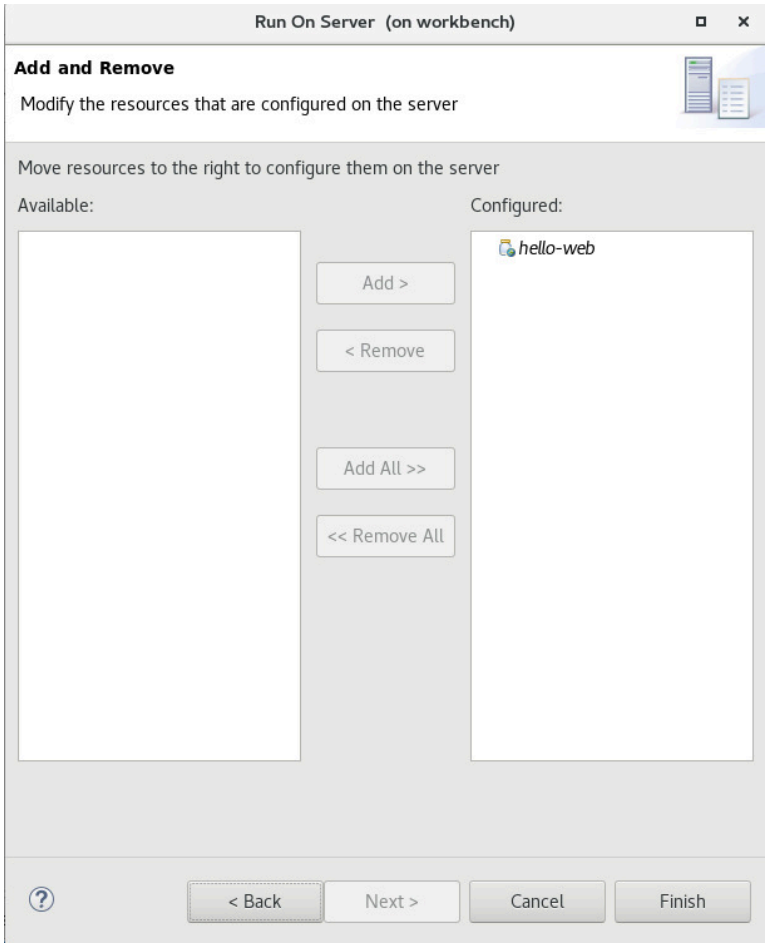


Figure 2.17: Deploy hello-web to JBoss EAP

6.4. Click **Console** in the IDE and observe the application being deployed. When the deployment is complete the **`hello-web`** application launches within the embedded

browser in the IDE. You can also open an external web browser and test the application just like you did in the previous step.

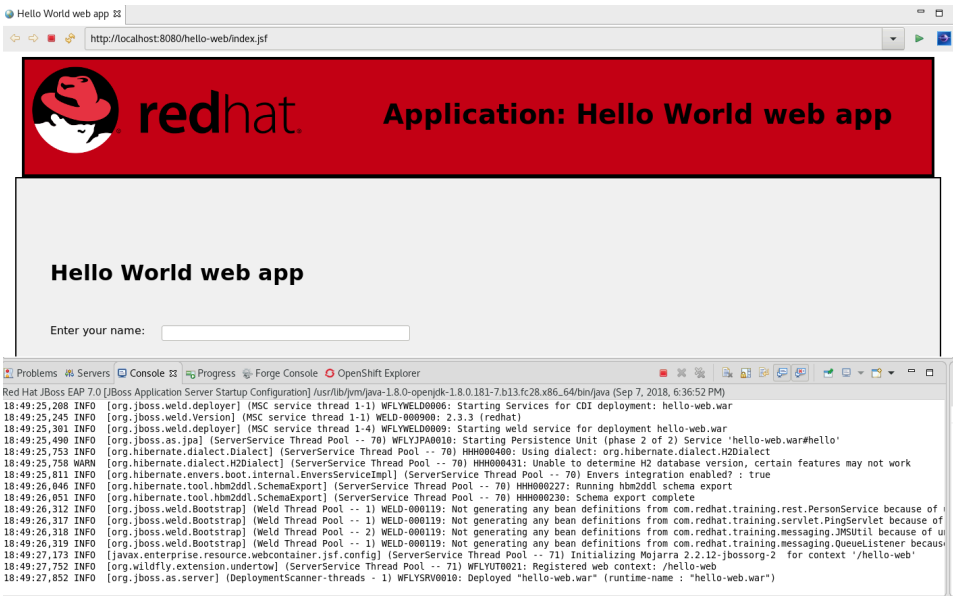


Figure 2.18: hello-web application deployed and running

6.5. To undeploy the **`hello-web`** application from within the IDE, expand the Red Hat JBoss EAP 7.0 item in the Servers tab at the bottom of the IDE. Right-click the **`hello-web`** entry, and select Remove to undeploy the application.

7. Right-click the **`hello-web`** project in the Project Explorer pane, and select Close Project to close this project.

8. 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.

SUMMARY

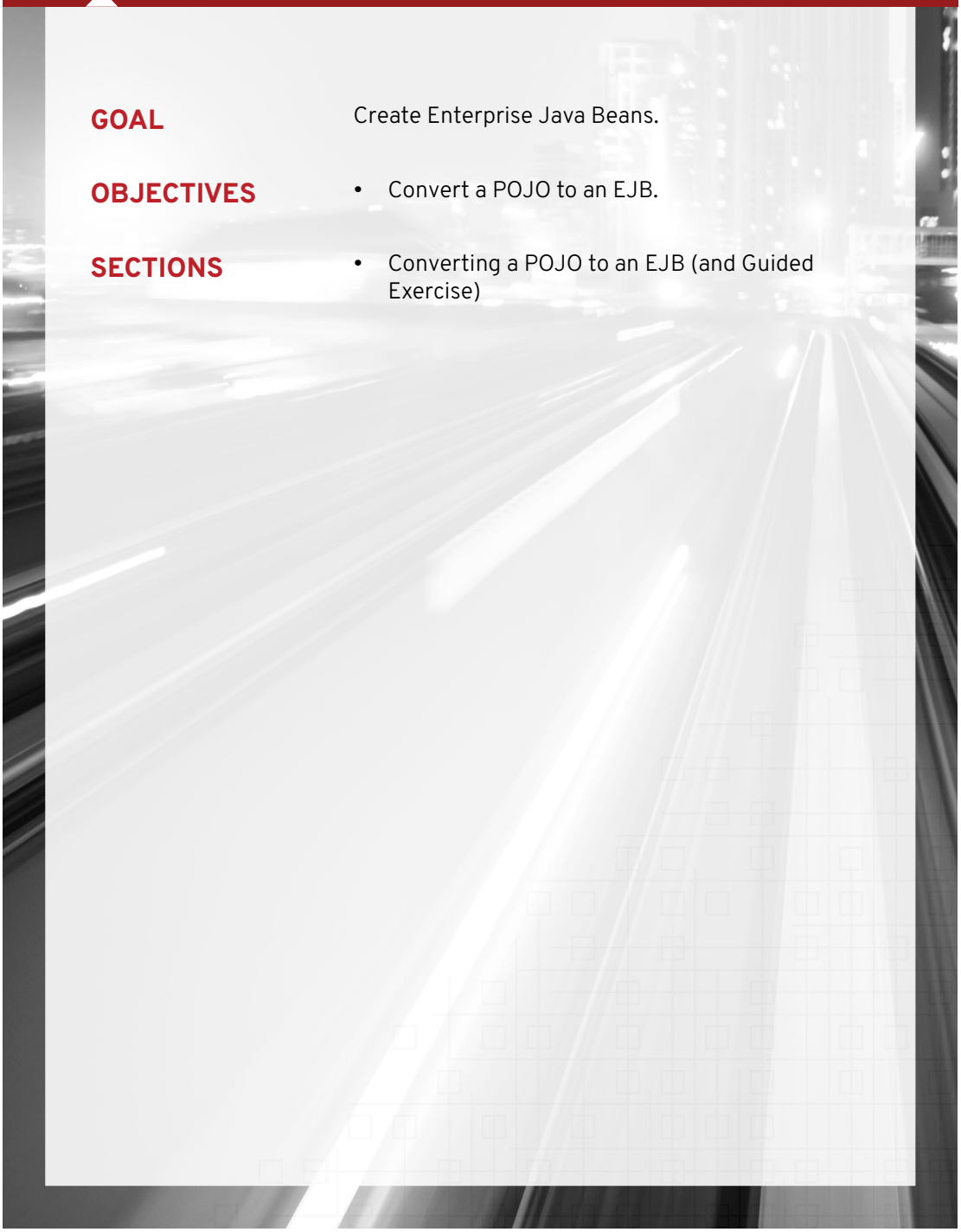
In this chapter, you learned:

- 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.
- 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

GOAL	Create Enterprise Java Beans.
OBJECTIVES	• Convert a POJO to an EJB.
SECTIONS	• Converting a POJO to an EJB (and Guided Exercise)



CONVERTING A POJO TO AN EJB

OBJECTIVE

After completing this section, students should be able to convert a Plain Old Java Object (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 end-points 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 RED HAT 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)**.

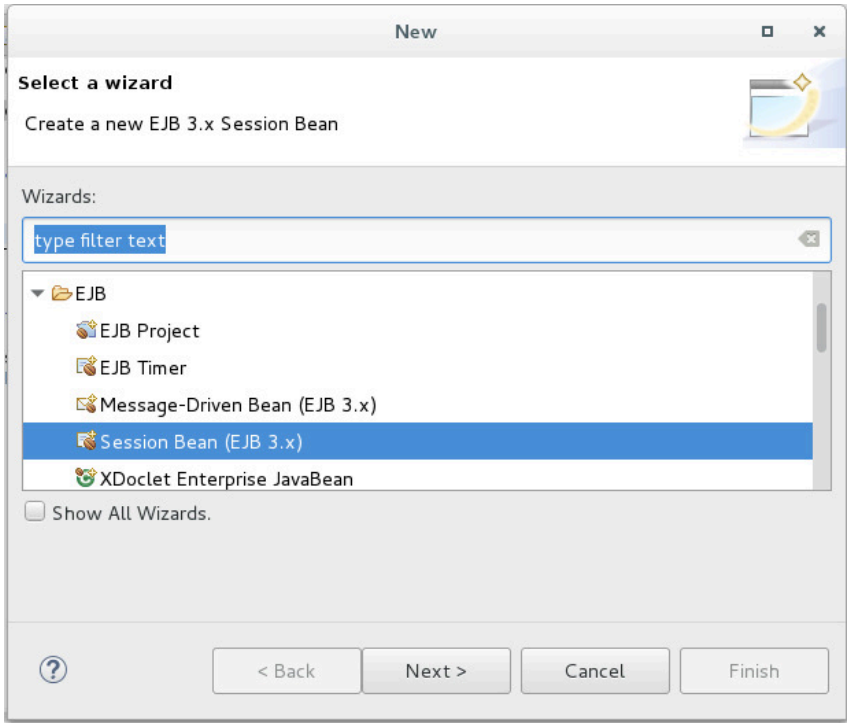


Figure 3.1: Create a new EJB in JBDS

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

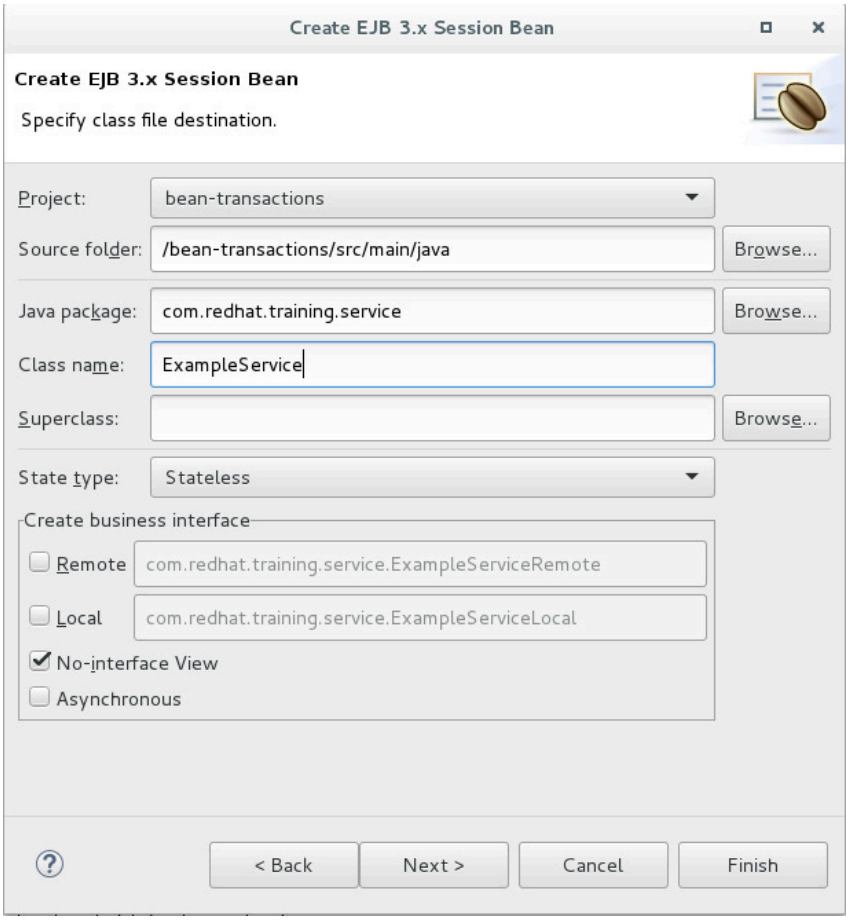


Figure 3.2: Name the new EJB Java class in JBDS and set its state

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**.

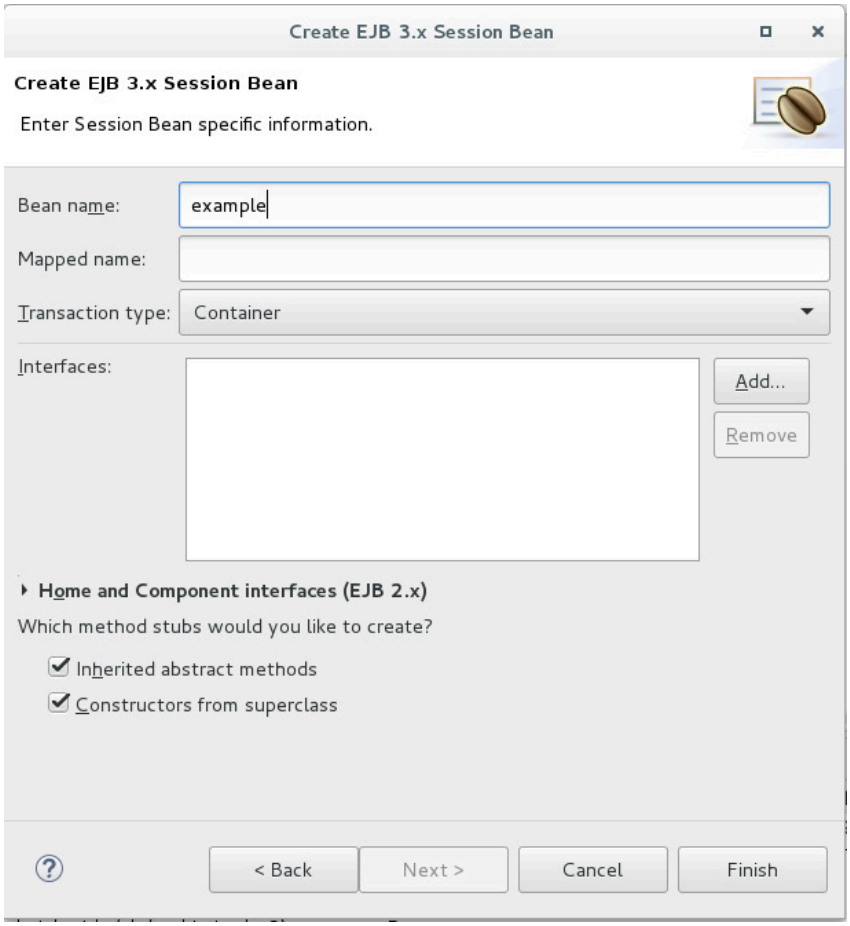


Figure 3.3: Provide a name for the EJB and setting its transaction type

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

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

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 and invoke a stateless EJB, and display the output 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

The source code for the application is available in a Git repository.

If you have not done so already, open a terminal window on your system, and run the following command to download the lab files required for this course.

```
$ git clone https://github.com/RedHatTraining/JB083x-lab
```

The above command creates a directory called **`JB083x-lab`**. This directory contains the source code for all the applications used in this course. There are two subdirectories in this directory named **`labs`** and **`solutions`**, which contain the source code for all the labs, and the corresponding solution files for the labs in this course.

The source code for the application used in this exercise is in the **`labs/stateless-ejb`** directory. The complete solution for this exercise is in the **`solutions/stateless-ejb`** directory.

1. Open Red Hat JBoss Developer Studio (JBDS) and import the Maven project.

1.1. Launch the Red Hat JBoss Developer Studio IDE.

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 **`JB083x-labs/labs/stateless-ejb`** directory, 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 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 are applied to the appropriate section of the **pom.xml** file.

   2.2. Click the Dependencies tab to view the project dependencies (the libraries, frameworks and modules that this project depends on).

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

   This also signals 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 when you submit the web form.

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

3.2. Explore the **Hello.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 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 the green "start" button to start the server. Watch the Console until the server starts and 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
```

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);
    }
}

...output omitted...
```

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. Press **`Ctrl`**+**`S`** to save the changes.

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.

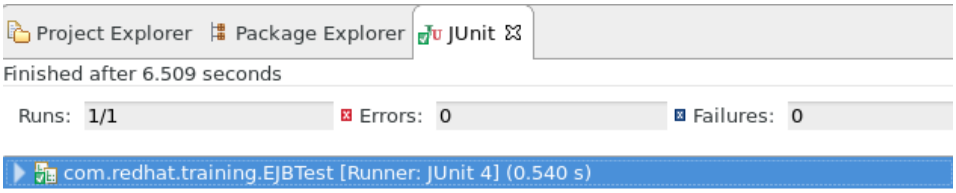


Figure 3.5: JUnit test success

8. Deploy the application to JBoss EAP using Maven by running the following commands:

```
$ cd JB083x-lab/labs/stateless-ejb
$ mvn clean wildfly:deploy
```

When the deployment is 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 application 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: `http://localhost:8080/stateless-ejb`.

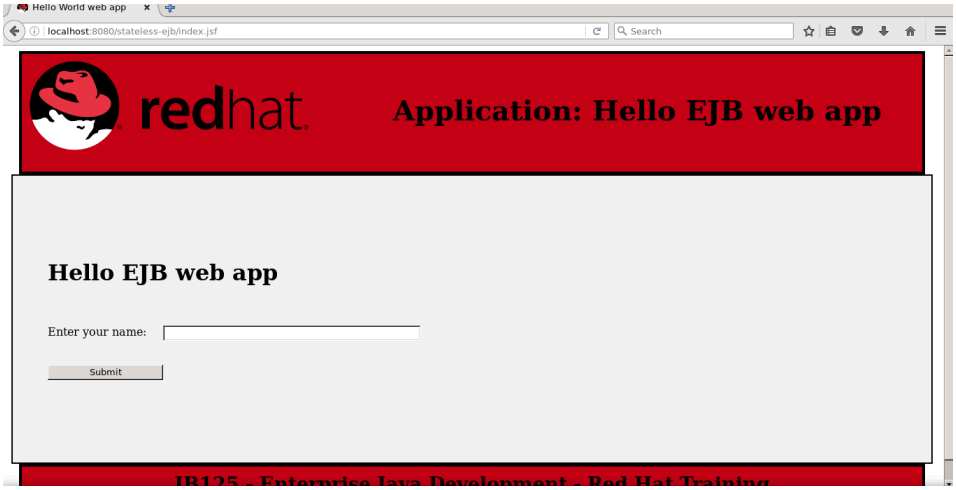


Figure 3.6: Application home page

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

The page updates with the message **`Hello Shadowman`**:

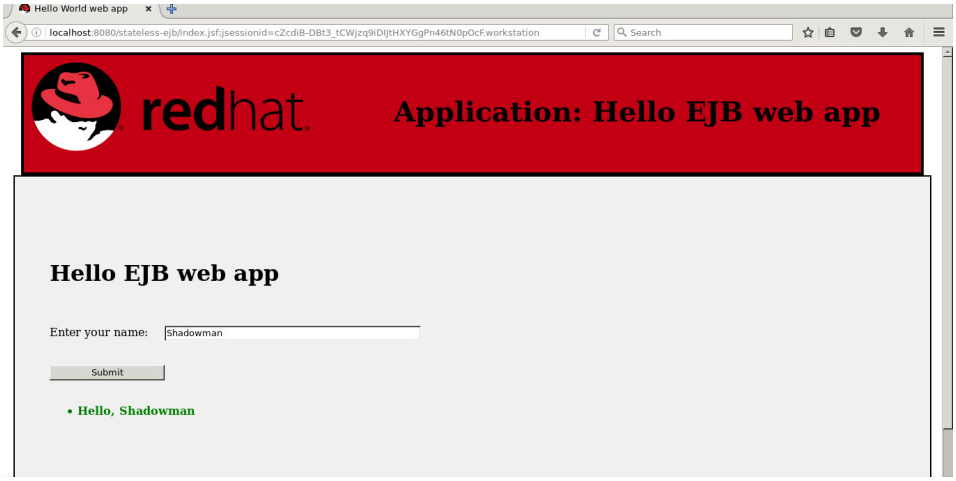


Figure 3.7: Application response

10. Undeploy the application and stop EAP.

10.1. Run the following command to undeploy the application:

```
$ mvn clean wildfly:undeploy
```

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

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

This concludes the guided exercise.

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.

CHAPTER 4

MANAGING PERSISTENCE

GOAL

Create Persistence Entities with validations.

OBJECTIVES

- Describe the Persistence API.
- Persist data to a data store using entities.
- Create a query using the Java Persistence Query Language.

SECTIONS

- Describing the Persistence API (and Quiz)
- Persisting Data (and Guided Exercise)
- Creating Queries (and Guided Exercise)

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.

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.

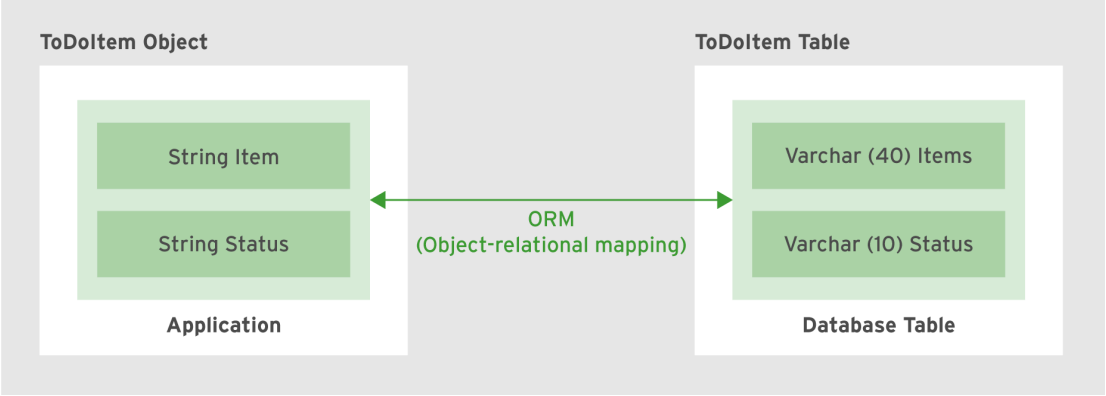


Figure 4.1: Impedance mismatch

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.

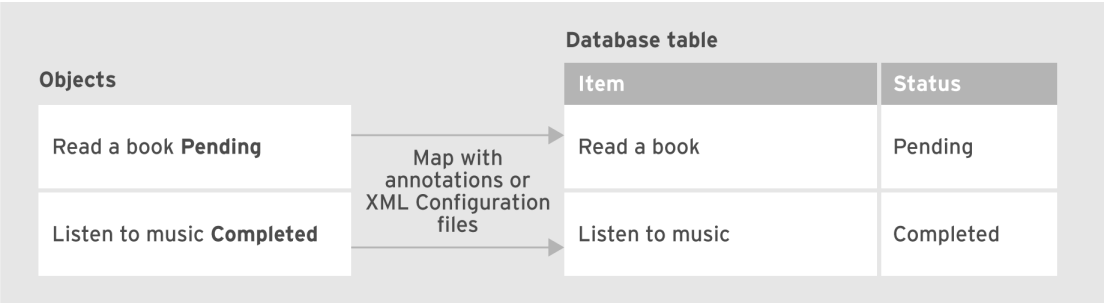


Figure 4.2: Object-relational mapping

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 persist-able. 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>
```

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 contex 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.

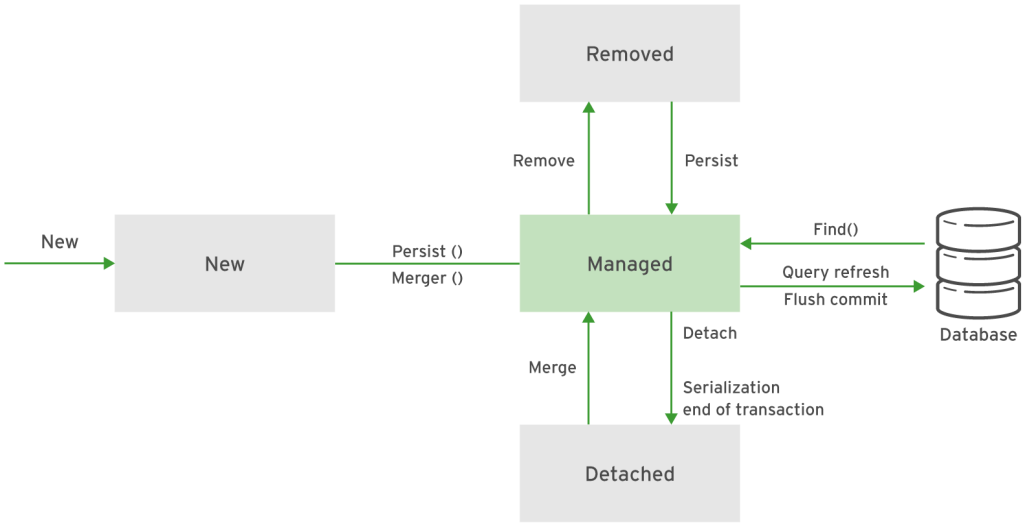


Figure 4.3: JPA components relationship

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

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 a database.

OUTCOMES

You should be able to create an entity class and persist entity data.

BEFORE YOU BEGIN

The source code for the application is available in a Git repository.

If you have not done so already, open a terminal window on your system, and run the following command to download the lab files required for this course.

```
$ git clone https://github.com/RedHatTraining/JB083x-lab
```

The above command creates a directory called **`JB083x-lab`**. This directory contains the source code for all the applications used in this course. There are two subdirectories in this directory named **`labs`** and **`solutions`**, which contain the source code for all the labs, and the corresponding solution files for the labs in this course.

The source code for the application used in this exercise is in the **`labs/persist-entity`** directory. The complete solution for this exercise is in the **`solutions/persist-entity`** directory.

1. Import the persist-entity project into the Red Hat JBoss Developer Studio IDE (JBDS).

1.1. Start the Red Hat JBoss Developer Studio IDE.

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 **`JB083x-lab/labs/persist-entity`** directory, 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.

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 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 {
   ...output omitted...
}
```

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 the **Serializable** interface. Import and implement the **Serializable** interface.

```
import javax.persistence.Entity;
import java.io.Serializable;

@Entity
public class Person implements Serializable {
    ...output omitted...
}
```

   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 a key generation strategy of **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;
    ...output omitted...
}
```

2.6. Press **Ctrl+S** to save the changes.

3. Open the **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;
   ...output omitted...
}
```

3.4. Persist a **Person** object 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) {
       ...output omitted...
 // call persist() method of entity manager to save the data
 entityManager.persist(p);
       ...output omitted...
}
```

3.5. Find the name of a Person object using the unique **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 name of the person 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. Press **Ctrl+S** to save your changes.

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 the **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 the green "start" button to start the server.

   5.2. Build and deploy the **persist-entity** application using the following commands in the terminal window:

```
$ cd JB083x-lab/labs/persist-entity
$ mvn clean wildfly:deploy
```

6. Test the application for persistence.

6.1. Use a web browser to navigate to `http://localhost:8080/persist-entity/` to access the **persist-entity** application.

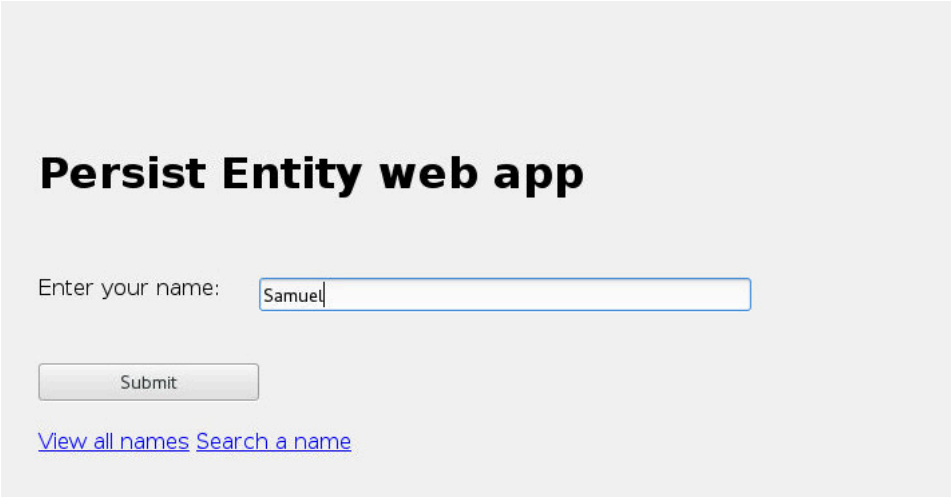


Figure 4.4: The persist-entity application

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.

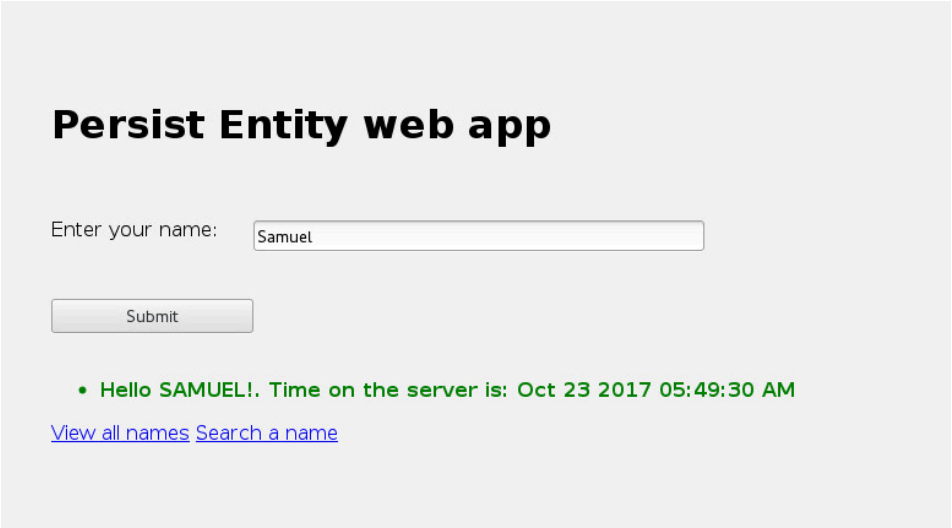


Figure 4.5: The persist-entity application response

6.4. Repeat the previous step at least two more times, entering different names to populate the database.

6.5. Click **View all names** to verify that all the names are stored in the database.

Figure 4.6: List of all names in the database

Click **Back** to return to the home page.

6.6. To find an individual person's name in the database, click the **Search a name** link.

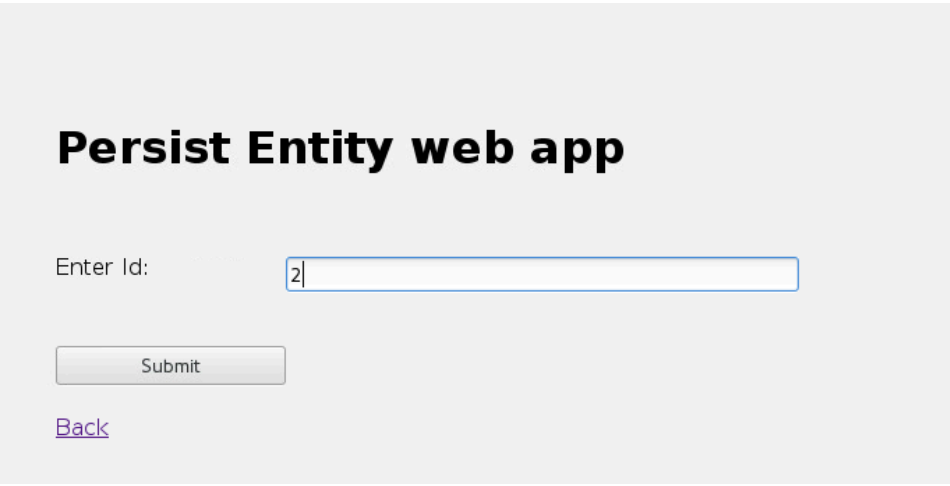


Figure 4.7: Finding an individual name in the database

6.7. Enter the value for an **`id`** in the **Enter Id:** field and click **Submit**.

Figure 4.8: Displaying an individual name from the database

Click **Back** to return 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:

```
$ mvn clean wildfly:undeploy
```

7.2. Right-click the **persist-entity** project in the **Project Explorer** pane, 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.

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 **`getResultList()`** 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.getResultList();
...
```

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.getResultList();
 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.

- 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);
 retrun 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

The source code for the application is available in a Git repository.

If you have not done so already, open a terminal window on your system and run the following command to download the lab files required for this course.

```
$ git clone https://github.com/RedHatTraining/JB083x-lab
```

The above command creates a directory called **`JB083x-lab`**. This directory contains the source code for all the applications used in this course. There are two subdirectories in this directory named **`labs`** and **`solutions`**, which contain the source code for all the labs, and the corresponding solution files for the labs in this course.

The source code for the application used in this exercise is in the **`labs/create-queries`** directory. The complete solution for this exercise is in the **`solutions/create-queries`** directory.

1. Import the create-queries project into the Red Hat JBoss Developer Studio IDE (JBDS).

1.1. Start the Red Hat JBoss Developer Studio IDE.

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 **`JB083x-lab/labs/create-queries`** directory, 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. Add a **getAllPersons()** method 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 **getAllPersons** method 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. Press **Ctrl**+**S** to save your changes.

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 in 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. Press **Ctrl**+**S** to save your changes.

4. Build and deploy the application.

   4.1. Start EAP by selecting the Servers tab in the bottom pane of JBDS. Right-click the Red Hat JBoss EAP 7.0 [Stopped] server and click the green "start" button to start the server.

   4.2. Build the create-queries application using the following commands in the terminal window:

```
$ cd JB083x-lab/labs/create-queries
$ mvn clean package
```

   4.3. Deploy the create-queries application using the following command in the terminal window:

```
$ mvn clean wildfly:deploy
```

5. Test the application to view all persons.
   5.1. Use a web browser 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.

Figure 4.9: 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 called **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. Press **Ctrl+S** to save your changes.

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. Press **Ctrl+S** to save your changes.

8. Use the following command in the terminal window to deploy the create-queries application:

```
$ mvn clean 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 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();
}
```

WARNING

Ensure you do not put quotes around **1** in the **setParameter** line. It is an **Integer** argument. If you quote the argument and make it a **String** argument, you will see the following error in the JBoss EAP logs when this query is executed:

```
11:09:58,076 WARN  [org.hibernate.jpa.spi.AbstractQueryImpl] (default task-25)
 HHH015014: DEPRECATION - attempt to refer to JPA positional parameter
 [?1] using String name ["1"] rather than int position [1] (generally in
 Query#setParameter, Query#getParameter or Query#getParameterValue calls).
```

   10.3. Press **Ctrl+S** to save your changes.

11. Use the following command in the terminal window to deploy the create-queries application:

```
$ mvn clean wildfly:deploy
```

12. Test the search feature that uses a positional parameter query.
   12.1. Use a web browser 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 the 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. Press **Ctrl**+**S** to save your changes.

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. Press **Ctrl**+**S** to save your changes.

14. Use the following command in the terminal window to deploy the create-queries application:

```
$ mvn clean wildfly:deploy
```

15. Test the search feature of the application using a named query.

15.1. Use a web browser 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:

```
$ mvn clean wildfly:undeploy
```

16.2. Right-click the **create-queries** project in the Project Explorer pane 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.

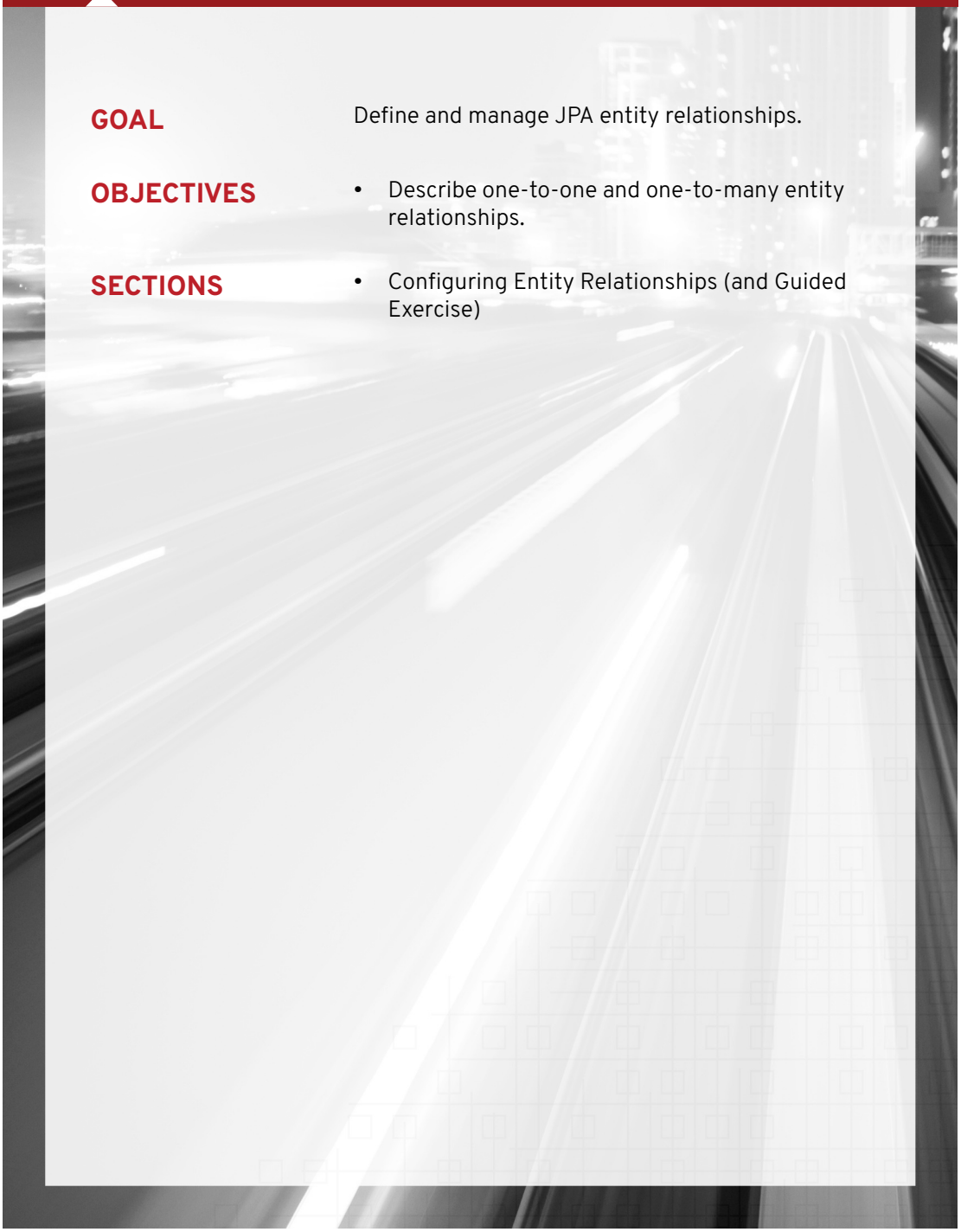
SUMMARY

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.
- 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



GOAL	Define and manage JPA entity relationships.
OBJECTIVES	• Describe one-to-one and one-to-many entity relationships.
SECTIONS	• Configuring Entity Relationships (and Guided Exercise)

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
@OneToOne	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.
@OneToMany	Defines an entity relationship as multi-valued, where one row in table X can be related to one or many rows in table Y.
@ManyToOne	Defines an entity relationship as multi-valued, where many rows in table X can be related to a single row in table Y.
@ManyToMany	Defines an entity relationship as multi-valued, where many rows in table X can be related to a one or may rows in table Y.
@JoinColumn	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)
```

```
@JoinColumn(name="userSSNID")(2)
private UserSSN userSSN;
```

(1) 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.
(2) 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")(1)
 private User user;
```

(1) 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;

 @ManyToOne(1)
 @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

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

The source code for the application is available in a Git repository.

If you have not done so already, open a terminal window on your system and run the following command to download the lab files required for this course.

```
$ git clone https://github.com/RedHatTraining/JB083x-lab
```

The above command creates a directory called **`JB083x-lab`**. This directory contains the source code for all the applications used in this course. There are two subdirectories in this directory named **`labs`** and **`solutions`**, which contain the source code for all the labs, and the corresponding solution files for the labs in this course.

The source code for the application used in this exercise is in the **`labs/entity-relationships`** directory. The complete solution for this exercise is in the **`solutions/entity-relationships`** directory.

1. Open Red Hat JBoss Developer Studio (JBDS) and import the Maven project.

1.1. Open Red Hat JBoss Developer Studio.

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/entity-relationships`** directory, 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, and then click 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. Press **Ctrl+S** to save your changes.

2.3. Open the **User** class by expanding the entity-relationships item in the Project Explorer tab in the left pane of JBDS, and then click 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. Press **Ctrl+S** to save your changes.

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, and then click 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. Press **Ctrl+S** to save your changes.

3.3. Open the **User** class by expanding the entity-relationships item in the Project Explorer tab in the left pane of JBDS, and then click 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. Press **Ctrl**+**S** to save your changes.

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:

```
$ cd JB083x-lab/labs/entity-relationships
$ mvn clean wildfly:deploy
```

When the deployment is complete, the output displays **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 the application is deployed, the following message 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. Navigate to `http://localhost:8080/entity-relationships` in a browser.

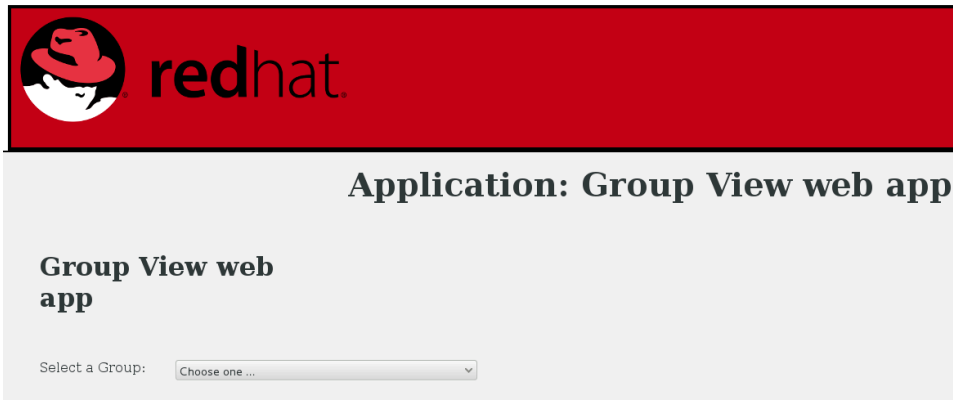


Figure 5.1: Application home page

6.2. Choose a group from the drop-down list to view the members in that group.

The page updates with a stack trace because of a **`LazyInitializationException`**:

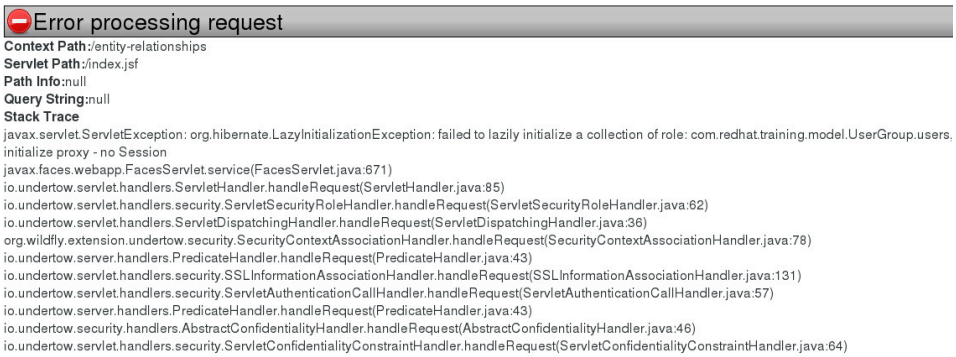


Figure 5.2: Application error response

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 user entity objects were not loaded into memory. Therefore, the JPA session was already closed when the JSF page attempted the call to **`getUsers()`**, causing the **`LazyInitializationException`**.

This error can be fixed by eagerly loading the user objects when fetching the group objects.

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, and 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. Press **Ctrl**+**S** to save your changes.

8. Redeploy the application on JBoss EAP using Maven by running the following commands:

```
$ mvn clean wildfly:deploy
```

When the deployment is complete, you should see **BUILD SUCCESS** in the output 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 application has been deployed successfully, the following 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.

9.2. Choose a group and view its users. The application displays users and email addresses of the group.

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 **`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. Press **`Ctrl`**+**`S`** to save your changes.

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. Press **Ctrl+S** to save your changes.

11. Redeploy the application on JBoss EAP using Maven by running the following commands:

```
$ mvn clean wildfly:deploy
```

When the deployment is complete, you should see **BUILD SUCCESS** in the output 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 application has been deployed successfully, the following message 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.

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:

```
$ mvn clean wildfly:undeploy
```

13.2. To close the project, right-click the **entity-relationships** project in the Project Explorer tab, 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.

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`**.

CHAPTER 6

CREATING REST SERVICES

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.

SECTIONS

- Describing Web Services Concepts (and Quiz)
- Creating REST Services with JAX-RS (and Guided Exercise)

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:

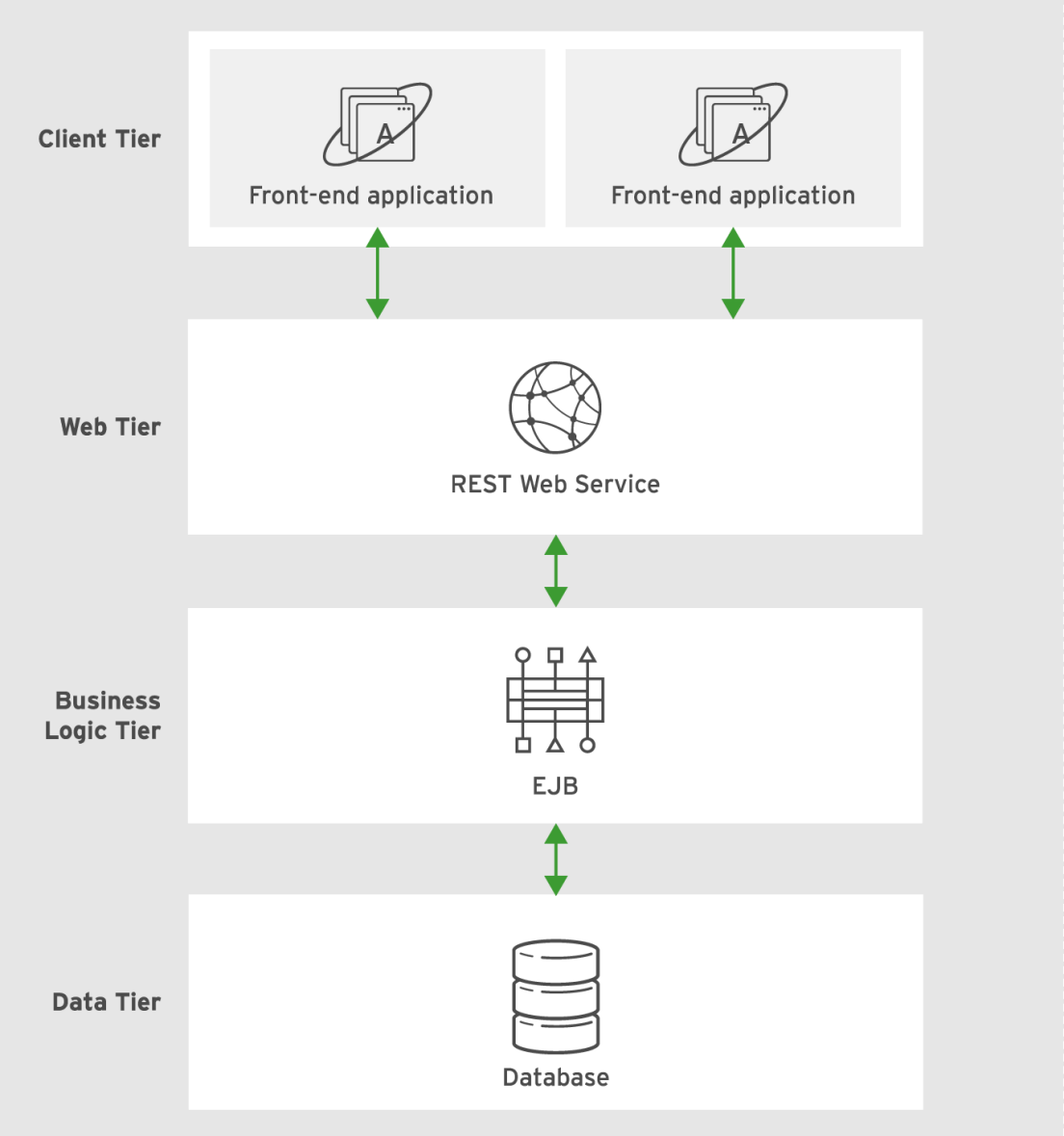


Figure 6.1: Web service application architecture

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/

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.

A Java EE application can utilize a RESTful web service to abstract the HTML or mobile 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 JSON format to make it easier to parse the data. For example, the following illustrates an HTTP GET request on the To Do List application's **`/api/items/{id}`** REST endpoint, which returns a To Do List item from the database based on the item's ID:

```
$ 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 **`hello-world`** 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 **`http://localhost:8080/hello-web/api/hello/1`**.
`@QueryParam`	The **`@QueryParam`** annotation is used to retrieve a parameter passed in through the URI as a query parameter, such as **`http://localhost:8080/hello-web/api/hello?id=1`**.

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 "<b>Hello World!</b>";
  }

  @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 {

  @GET(1)
  @Path("person")
  public List<Person> getPersons() {
    ...implementation omitted...
  }

  @POST(2)
  @Path("person")
  public String savePerson(Person person) {
     ...implementation omitted...
  }

  @PUT(3)
  @Path("person")
  public String updatePerson(Person person) {
     ...implementation omitted...
  }

  @DELETE(4)
  @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 URI: `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 URI: `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 URI: `http://localhost:8080/hello-world/hello/person`.
4. This method deletes a **`Person`** object when an HTTP DELETE request is made to the following URI: `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 sendEmail(1)) {
 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 * 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/

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 the Firefox REST client add-on.

BEFORE YOU BEGIN

The source code for the application is available in a Git repository.

If you have not done so already, open a terminal window on your system and run the following command to download the lab files required for this course.

```
$ git clone https://github.com/RedHatTraining/JB083x-lab
```

The above command creates a directory called **`JB083x-lab`**. This directory contains the source code for all the applications used in this course. There are two subdirectories in this directory named **`labs`** and **`solutions`**, which contain the source code for all the labs, and the corresponding solution files for the labs in this course.

The source code for the application used in this exercise is in the **`labs/hello-rest`** directory. The complete solution for this exercise is in the **`solutions/hello-rest`** directory.

1. Import the hello-rest project into the Red Hat JBoss Developer Studio (JBDS) IDE.

1.1. Start the Red Hat JBoss Developer Studio IDE.

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 **`JB083x-lab/labs/hello-rest`** directory 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. 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`** and then 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. Press **`Ctrl`**+**`S`** to save your changes.

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 endpoints 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 object 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`** object 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`** object, 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 object 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 {
   ...output omitted...
 }
```

A POST request to `http://localhost:8080/hello-rest/api/persons/` with a JSON representation of a **Person** object now persists that person to the database.

6.7. Press **Ctrl**+**S** to save your changes.

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 "start" button to start the server.

8. Deploy the hello-rest application using the following commands in the terminal window:

```
$ cd JB083x-lab/labs/hello-rest
$ mvn clean wildfly:deploy
```

9. Test the REST API using the Firefox REST Client add-on.

9.1. Start Firefox and click the REST Client add-on in the browser's toolbar.

Figure 6.2: The Firefox REST client add-on

9.2. In the top toolbar, click **Headers** and then 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`**

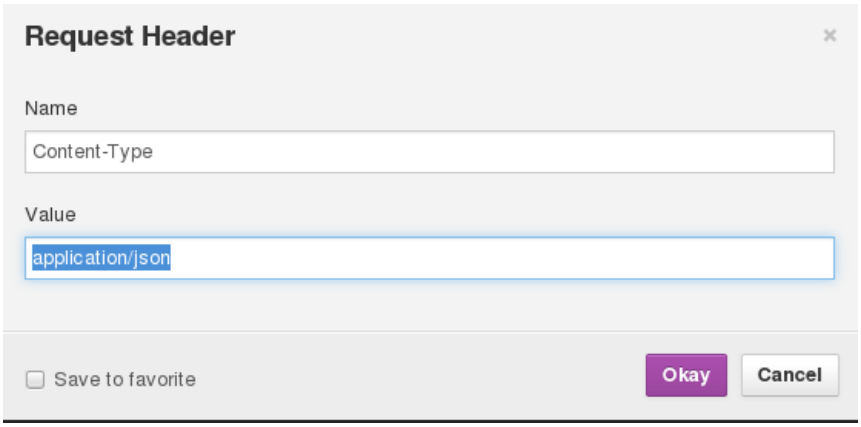


Figure 6.3: Creating a custom request header in the REST Client

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`**.

Figure 6.4: Response to the HTTP POST request

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. List all the **Person** objects.

Select GET as the Method. In the URL form, enter **http://localhost:8080/hello-rest/api/persons** and then click Send.

Verify in the Response Headers tab that the Status Code is **200 OK**, and in the Response tab, the body should contain the following:

```
[{"id":1,"name":"Shadowman"}]
```

Figure 6.5: Response to the HTTP GET request

9.9. Delete the newly created **Person** object by ID.

Select DELETE as the Method. In the URL form, enter **http://localhost:8080/hello-rest/api/persons/1** and then click Send.

Verify in the Response Headers tab that the Status Code is **204**.

Repeat the step to list all the **Person** objects using a GET request and verify that the response body shows an empty array:

```
[]
```

10. Undeploy the application and stop EAP.

10.1. In the terminal window, run the following command to undeploy the application from EAP:

```
$ mvn clean wildfly:undeploy
```

10.2. Right-click the **hello-rest** project in the Project Explorer tab, 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.

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.