Java EE 7 Hands-on Lab

Arun Gupta

Revision 2.0
Jan 28, 2014

Table of Contents

List of Figures

Chapter 1. Introduction

The Java EE 7 platform continues the ease of development push that characterized prior releases by bringing further simplification to enterprise development. It adds new and important APIs such as the REST client API in JAX-RS 2.0 and the long awaited Batch Processing API. Java Message Service 2.0 has undergone an extreme makeover to align with the improvements in the Java language. There are plenty of improvements to several other components. Newer web standards like HTML 5, WebSocket, and JSON processing are embraced to build modern web applications.

This hands-on lab will build a typical 3-tier end-to-end application using the following Java EE 7 technologies:

- Java API for WebSocket 1.0 (JSR 356)
- Batch Applications for the Java Platform 1.0 (JSR 352)
- Java API for JSON Processing 1.0 (JSR 353)
- Java API for RESTful Web Services 2.0 (JSR 339)
- Java Message Service 2.0 (JSR 343)
- Java Persistence API 2.1 (JSR 338)
- JavaServer Faces 2.2 (JSR 344)
- Contexts and Dependency Injection 1.1 (JSR 346)
- Bean Validation 1.1 (JSR 349)
- Java Transaction API 1.2 (JSR 907)

Together these APIs will allow you to be more productive by simplifying enterprise development.

The latest version of this document can be downloaded from javaee7-hol.html[1].

1.1. Software Requirement

The following software needs to be downloaded and installed:

- JDK 7 from http://www.oracle.com/technetwork/java/javase/downloads/index.html.

[1] https://github.com/javaee-samples/javaee7-hol/blob/master/docs/asciidoc/javaee7-hol.html

- **Application Server**: This lab can use WildFly 8 or GlassFish 4 as the application server. This document provide instructions for WildFly 8.
- **IDE**: NetBeans 7.4+, JBoss Developer Studio (Eclipse-based), or IntelliJ IDEA 13 can be used. This document provide instructions for NetBeans 8.
  Download "All" or "Java EE" version from http://netbeans.org/downloads/. A snapshot of the downloads page is shown and highlights the exact 'Download' button to be clicked.

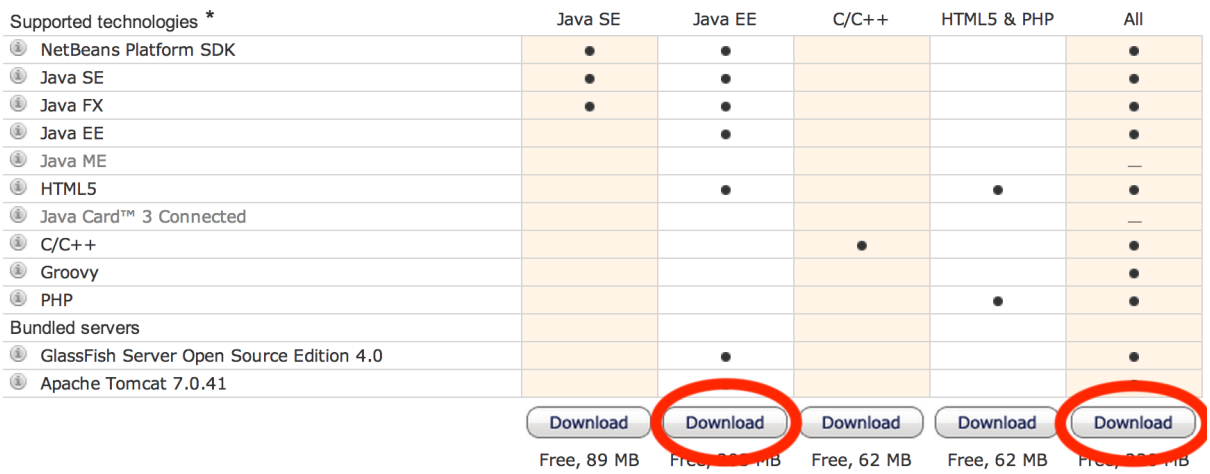


Figure 1.1.

WildFly 8 needs to be downloaded from wildfly.org[2] and configured in NetBeans IDE following the instructions in Section A.1, "Configure WildFly 8 in NetBeans".

Section A.2, "Prepare IntelliJ IDEA for working with WildFly 8" explains how to configure WildFly in IntelliJ IDEA.

[2] http://wildfly.org/downloads/

Chapter 2. Problem Statement

This hands-on lab builds a typical 3-tier Java EE 7 Web application that allows customers to view the show timings for a movie in a 7-theater Cineplex and make reservations. Users can add new movies and delete existing movies. Customers can discuss the movie in a chat room. Total sales from each showing are calculated at the end of the day. Customers also accrue points for watching movies.

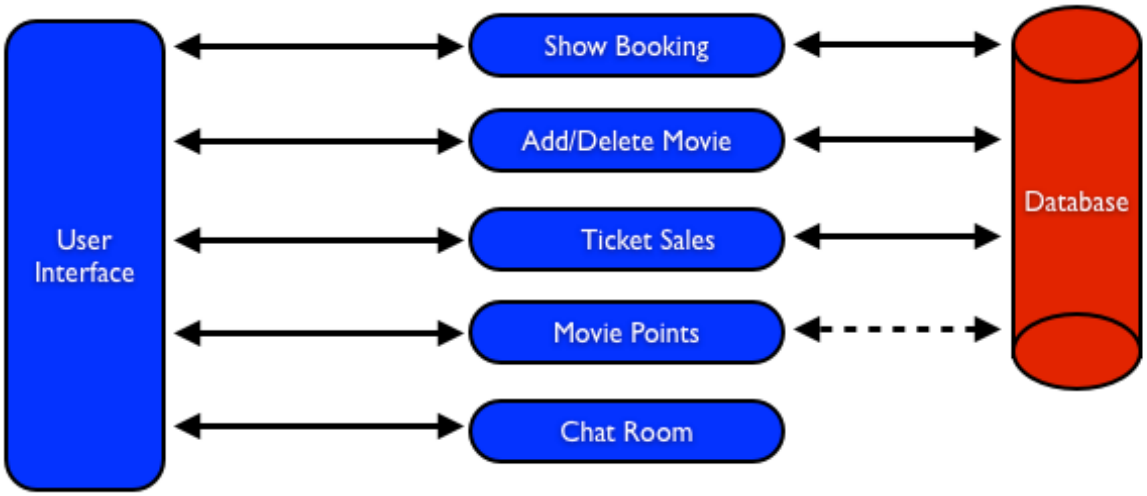


Figure 2.1.

This figure shows the key components of the application. The User Interface initiates all the flows in the application. Show Booking, Add/Delete Movie and Ticket Sales interact with the database; Movie Points may interact with the database, however, this is out of scope for this application; and Chat Room does not interact with the database.

The different functions of the application, as detailed above, utilize various Java technologies and web standards in their implementation. The following figure shows how Java EE technologies are used in different flows.

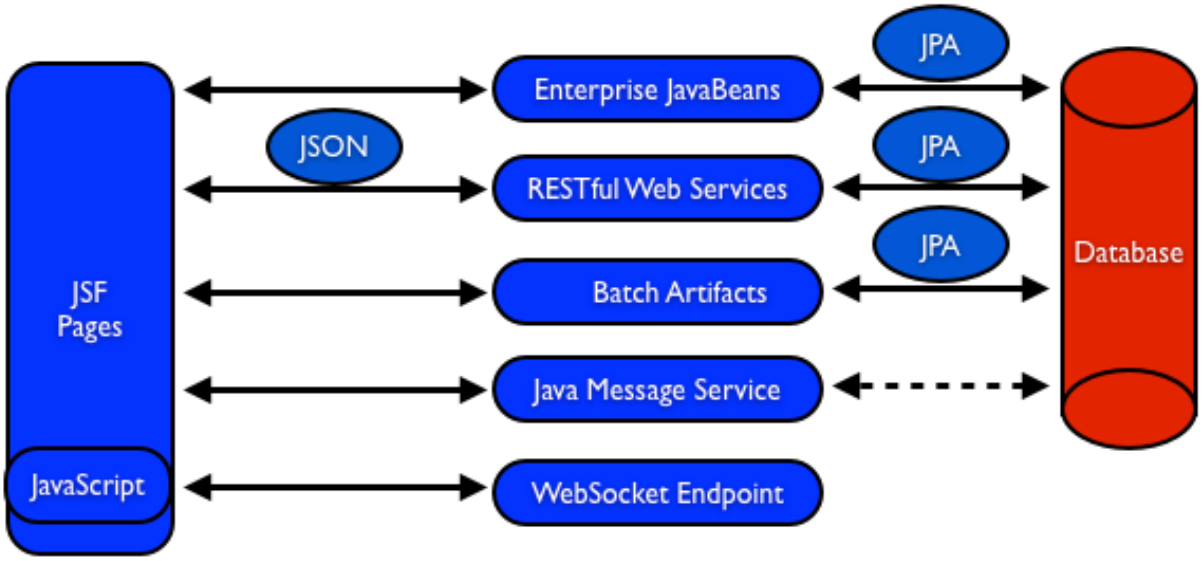


Figure 2.2.

The table below details the components and the selected technology used in its' implementation.

Flow	Description
User Interface	Written entirely in *JavaServer Faces* (JSF)
Chat Room	Utilizes client-side JavaScript and JSON to communicate with a *WebSocket* endpoint
Ticket Sales	Uses *Batch Applications for the Java Platform* to calculate the total sales and persist to the database.
Add/Delete Movie	Implemented using RESTful Web Services. JSON is used as on-the-wire data format
Movie Points	Uses *Java Message Service* (JMS) to update and obtain loyalty reward points; an optional implementation using database technology may be performed
Show Booking	Uses lightweight *Enterprise JavaBeans* to communicate with the database using Java Persistence API

This document is not a comprehensive tutorial of Java EE. The attendees are expected to know the basic Java EE concepts such as EJB, JPA, JAX-RS, and CDI. The Java EE 7 Tutorial[1] is a good place to learn all these concepts. However enough explanation is provided in this guide to get you started with the application.

[1] http://docs.oracle.com/javaee/7/tutorial/doc/

This is a sample application and the code may not be following the best practices to prevent SQL injection, cross-side scripting attacks, escaping parameters, and other similar features expected of a robust enterprise application. This is intentional such as to stay focused on explaining the technology. It is highly recommended to make sure that the code copied from this sample application is updated to meet those requirements.

2.1. Lab Flow

The attendees will start with an existing maven application and by following the instructions and guidance provided by this lab they will:

- Read existing source code to gain an understanding of the structure of the application and use of the selected platform technologies.
- Add new and update existing code with provided fragments in order to demonstrate usage of different technology stacks in the Java EE 7 platform.

While you are copy/pasting the code from this document into NetBeans, here are couple of tips that will be really useful and make your experience enjoyable!

- NetBeans provides capability to neatly format the source code following conventions. This can be done for any type of source code, whether its XML or Java or something else. It is highly recommended to use this functionality after the code is copy/pasted from this document to the editor. This keeps the code legible.

This functionality can be accessed by right-clicking in the editor pane and selecting “Format” as shown.

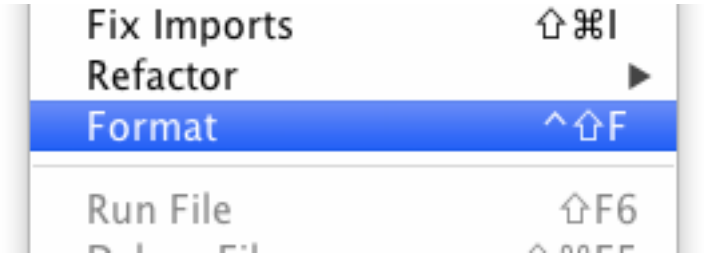


Figure 2.3.

This functionality is also accessible using the following keyboard shortcuts:

Shortcut	Operating System
Ctrl + Shift + F	Mac

Shortcut	Operating System
Alt + Shift + F	Windows
Alt + Shift + F	Linux

- Copy/pasting the Java code from this document in NetBeans editor does not auto-import the classes. This is required to be done manually in order for the classes to compile. This can be fixed for each missing import statement by clicking on the yellow bulb shown in the side bar.

Figure 2.4.

Alternatively all the imports can be resolved by right-clicking on the editor pane and selecting “Fix Imports” as shown.

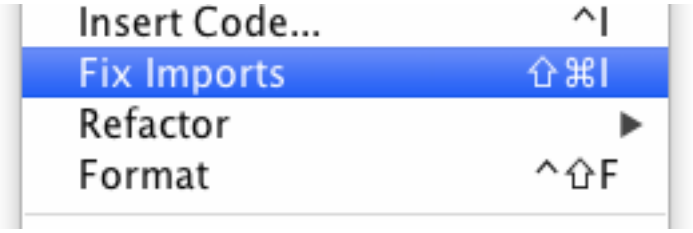


Figure 2.5.

This functionality is also accessible using the following keyboard shortcuts:

Shortcut	Operating System
Command + Shift + I	Mac
Ctrl + Shift + I	Windows
Ctrl + Shift + I	Linux

The defaults may work in most of the cases. Choices are shown in case a class is available to import from multiple packages. If multiple packages are available then specific packages to import from are clearly marked in the document.

2.2. Estimated Time

Following the complete instructions in this document can take any where from two to four hours. The wide time range accommodates for learning the new technologies, finding your way in NetBeans, copy/pasting the code, and debugging the errors.

The recommended flow is where you follow through the instructions in all sections in the listed sequence. Alternatively, you may like to cover section Chapter 3, *Walk-through of Sample Application* through Chapter 9, *Show Booking (JavaServer Faces)* in an order of your choice, based upon your interest and preference of the technology. However section Chapter 6, *View and Delete Movie (Java API for RESTful Web Services)* is a pre-requisite for Chapter 7, *Add Movie (Java API for JSON Processing)*.

Here is an approximate time estimate for each section:

Section Title	Estimated Time
Chapter 3, *Walk-through of Sample Application*	15 - 30 mins
Chapter 4, *Chat Room (Java API for WebSocket)*	30 - 45 mins
Chapter 5, *Ticket Sales (Batch Applications for the Java Platform)*	30 - 45 mins
Chapter 6, *View and Delete Movie (Java API for RESTful Web Services)*	30 - 45 mins
Chapter 7, *Add Movie (Java API for JSON Processing)*	30 - 45 mins
Chapter 8, *Movie Points (Java Message Service)*	30 - 45 mins
Chapter 9, *Show Booking (JavaServer Faces)*	30 - 45 mins

The listed time for each section is only an estimate and by no means restrict you within that. These sections have been completed in much shorter time, and you can do it too!

The listed time for each section also allows you to create a custom version of the lab depending upon your target audience and available time.

Chapter 3. Walk-through of Sample Application

Purpose: This section will download the sample application to be used in this hands-on lab. A walk-through of the application will be performed to provide an understanding of the application architecture.

Estimated Time: 15-30 mins

1. Download the sample application from movieplex7-starting-template.zip[1] and unzip. This will create a 'movieplex7' directory and unzips all the content there.
2. In NetBeans IDE, select 'File', 'Open Project', select the unzipped directory, and click on 'Open Project'. The project structure is shown.

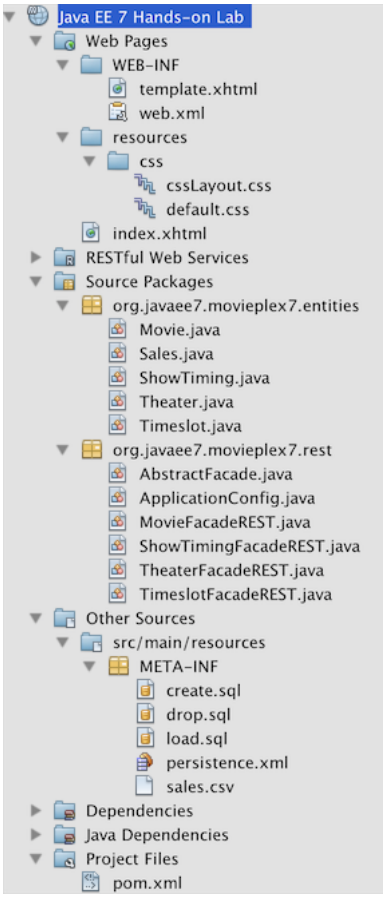


Figure 3.1.

[1] https://github.com/javaee-samples/javaee7-hol/blob/master/starting-template/movieplex7-starting-template.zip?raw=true

3. Maven Coordinates: Expand ‘Project Files’ and double click on ‘pom.xml’. In the ‘pom.xml’, the Java EE 7 API is specified as a <dependency>:

```
<dependencies>
    <dependency>
        <groupId>javax</groupId>
        <artifactId>javaee-api</artifactId>
        <version>7.0</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

This will ensure that Java EE 7 APIs are retrieved from the central Maven repository.

The Java EE 6 platform introduced the notion of ‘profiles’. A profile is a configuration of the Java EE platform targeted at a specific class of applications. All Java EE profiles share a set of common features, such as naming and resource injection, packaging rules, security requirements, etc. A profile may contain a proper subset or superset of the technologies contained in the platform.

The Java EE Web Profile is a profile of the Java EE Platform specifically targeted at modern web applications. The complete set of specifications defined in the Web Profile is defined in the Java EE 7 Web Profile Specification.

WildFly can be started in Full Platform or Web Profile.

This lab requires Full Platform download. All technologies used in this lab, except Java Message Service and Batch Applications for the Java Platform, can be deployed on Web Profile.

4. **Default Data Source**: Expand ‘Other Sources’, ‘src/main/resources’, ‘META-INF’, and double-click on ‘persistence.xml’. By default, NetBeans opens the file in Design View. Click on ‘Source’ tab to view the XML source.

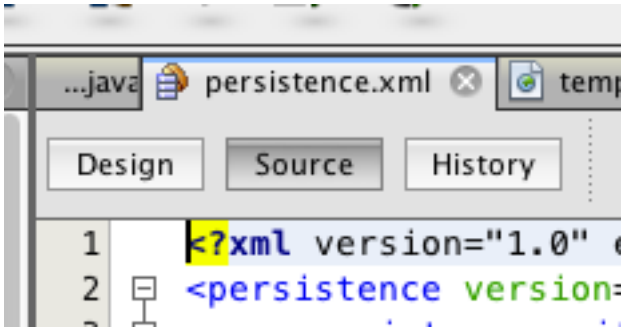


Figure 3.2.

It looks like:

```
<?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="movieplex7PU" transaction-type="JTA">
        <!--
            <jta-data-source>java:comp/DefaultDataSource</jta-data-
source>
        -->
        <properties>
            <property
                name="javax.persistence.schema-
generation.database.action"
                value="drop-and-create"/>
            <property
                name="javax.persistence.schema-generation.create-
source"
                value="script"/>
            <property
                name="javax.persistence.schema-generation.drop-source"
                value="script"/>
            <property
                name="javax.persistence.schema-generation.drop-script-
source"
                value="META-INF/drop.sql"/>
            <property
                name="javax.persistence.sql-load-script-source"
                value="META-INF/load.sql"/>
            <property
```

```
                name="eclipselink.deploy-on-startup"
                value="true"/>
            <property
                name="eclipselink.logging.exceptions"
                value="false"/>
        </properties>
    </persistence-unit>
</persistence>
```

Notice `<jta-data-source>` is commented out, i.e. no data source element is specified. This element identifies the JDBC resource to connect to in the runtime environment of the underlying application server.

The Java EE 7 platform defines a new default data source that must be provided by the runtime. This pre-configured data source is accessible under the JNDI name

```
java:comp/DefaultDataSource
```

The JPA 2.1 specification says if neither `jta-data-source` nor `non-jta-data-source` elements are specified, the deployer must specify a JTA data source or the default JTA data source must be provided by the container.

For WildFly 8, the default data source is bound to the JDBC resource `what name`.

Clicking back and forth between ‘Design’ and ‘Source’ view may prompt the error shown below:

Figure 3.3.

This will get resolved when we run the application. Click on ‘OK’ to dismiss the dialog.

5. **Schema Generation:** JPA 2.1 defines a new set of `javax.persistence.schema-generation.*` properties that can be used to generate database artifacts like tables, indexes, and constraints in a database

schema. This helps in prototyping of your application where the required artifacts are generated either prior to application deployment or as part of `EntityManagerFactory` creation. This feature will allow your JPA domain object model to be directly generated in a database. The generated schema may need to be tuned for actual production environment.

The “persistence.xml” in the application has the following `javax.persistence.schema-generation.*` properties. Their meaning and possible values are explained:

Property	Meaning	Values
`javax.persistence.schema-generation.database.action`	Specifies the action to be taken by the persistence provider with regard to the database artifacts.	`none`, `create`, `drop-and-create`, `drop`
`javax.persistence.schema-generation.create-source` `javax.persistence.schema-generation.drop-source`	Specifies whether the creation or deletion of database artifacts is to occur on the basis of the object/relational mapping metadata, DDL script, or a combination of the two.	`metadata`, `script`, `metadata-then-script`, `script-then-metadata`
`javax.persistence.schema-generation.create-script-source` `javax.persistence.schema-generation.drop-script-source`	Specifies a `java.IO.Reader` configured for reading of the SQL script or a string designating a file URL for the SQL script to create or delete database artifacts.	
`javax.persistence.sql-load-script-source`	Specifies a `java.IO.Reader` configured for reading of the SQL load script for database initialization or a string designating a file URL for the script.	

Refer to the JPA 2.1 Specification[2] for a complete understanding of these properties.

In the application, the scripts are bundled in the WAR file in 'META-INF' directory. As the location of these scripts is specified as a URL, the scripts may be loaded from outside the WAR file as well.

Feel free to open 'create.sql', 'drop.sql' and 'load.sql' and read through the SQL scripts. The database schema is shown.

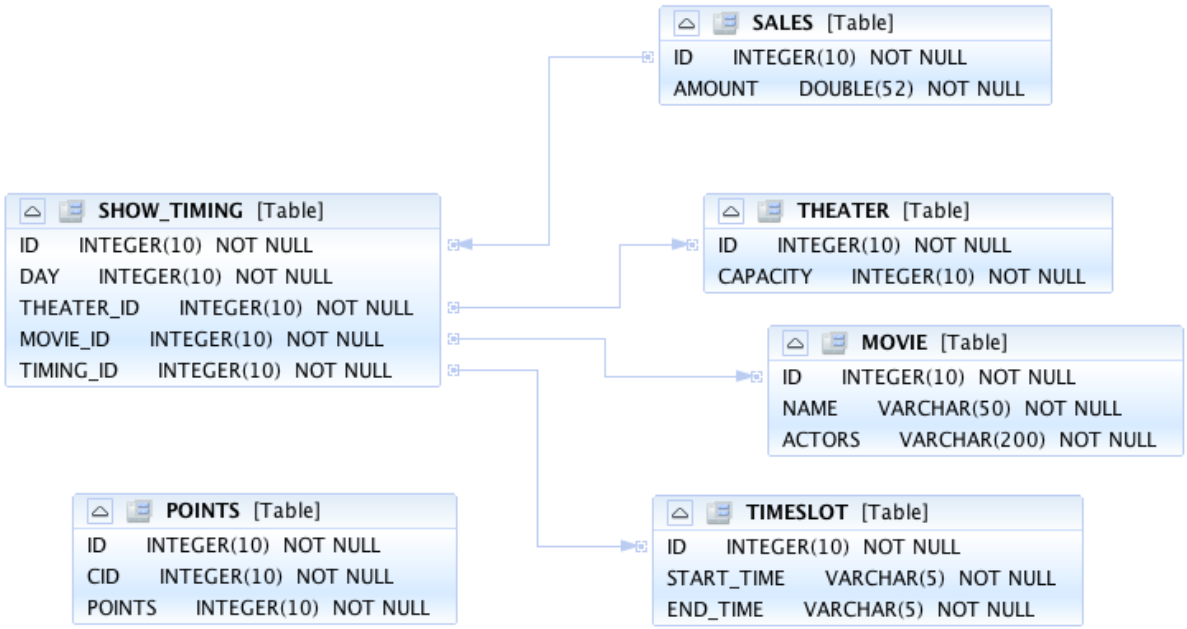


Figure 3.4.

This folder also contains 'sales.csv' which carries some comma-separated data, and is used later in the application.

6. **JPA entities, Stateless EJBs, and REST endpoints**: Expand `Source Packages'. The package `org.javaee7.movieplex7.entities` contains the JPA entities corresponding to the database table definitions. Each JPA entity has several convenient `@NamedQuery` defined and uses Bean Validation constraints to enforce validation.
   The package `org.javaee7.movieplex7.rest` contains stateless EJBs corresponding to different JPA entities.

[2] http://jcp.org/en/jsr/detail?id=338

Each EJB has methods to perform CRUD operations on the JPA entity and convenience query methods. Each EJB is also EL-injectable (@Named) and published as a REST endpoint (@Path). The AplicationConfig class defines the base path of REST endpoint. The path for the REST endpoint is the same as the JPA entity class name.

The mapping between JPA entity classes, EJB classes, and the URI of the corresponding REST endpoint is shown.

JPA Entity Class	EJB Class	RESTful Path
`Movie`	`MovieFacadeREST`	/webresources/movie
`Sales`	`SalesFacadeREST`	/webresources/sales
`ShowTiming`	`ShowTimingFacadeREST`	/webresources/showtiming
`Theater`	`TheaterFacadeREST`	/webresources/theater
`Timeslot`	`TimeslotFacadeREST`	/webresources/timeslot

Feel free to browse through the code.

7. **JSF pages**: 'WEB-INF/template.xhtml' defines the template of the web page and has a header, left navigation bar, and a main content section. 'index.xhtml' uses this template and the EJBs to display the number of movies and theaters.
   Java EE 7 enables CDI discovery of beans by default. No 'beans.xml' is required in 'WEB-INF'. This allows all beans with bean defining annotation, i.e. either a bean with an explicit CDI scope or EJBs to be available for injection.

   Note, 'template.xhtml' is in 'WEB-INF' folder as it allows the template to be accessible from the pages bundled with the application only. If it were bundled with rest of the pages then it would be accessible outside the application and thus allowing other external pages to use it as well.

8. **Run the sample**: Right-click on the project and select 'Run'. This will download all the maven dependencies on your machine, build a WAR file, deploy on WildFly 8 , and show the URL localhost:8080/movieplex7[3] in the default browser configured in NetBeans. Note that this could take a while if you have never built a Maven application on your machine.

[3] http://localhost:8080/movieplex7

The project will show red squiggly lines in the source code indicating that the classes cannot be resolved. This is expected before the dependencies are downloaded. However these references will be resolved correctly after the dependencies are downloaded during project building.

During the first run, the IDE will ask you to select a deployment server. Choose the configured WildFly server and click on 'OK'.

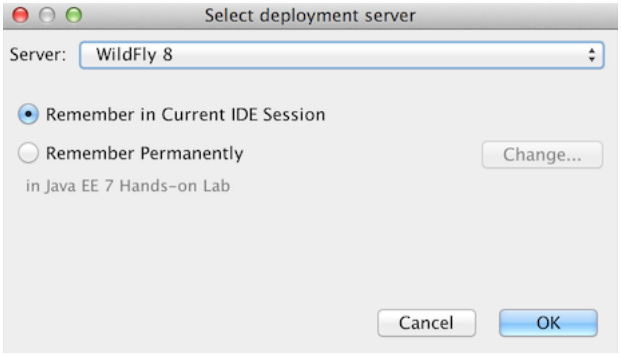


Figure 3.5.

The output looks like as shown.

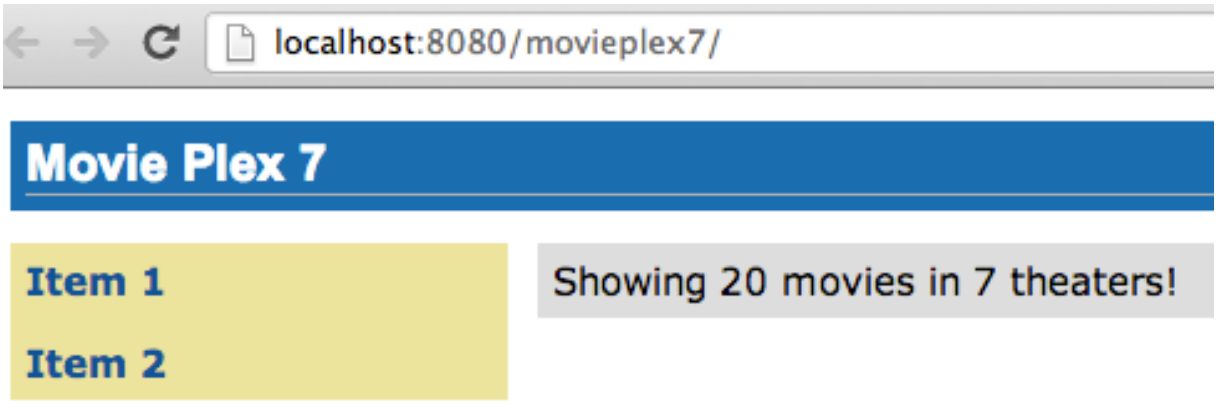


Figure 3.6.

Chapter 4. Chat Room (Java API for WebSocket)

Purpose: Build a chat room for viewers. In doing so several new features of Java API for WebSocket 1.0 will be introduced and demonstrated by using them in the application.

Estimated Time: 30-45 mins

WebSocket provide a full-duplex and bi-directional communication protocol over a single TCP connection. WebSocket is a combination of IETF RFC 6455[1] Protocol[2] and W3C JavaScript WebSocket API[3] (a Candidate Recommendation as of this writing). The protocol defines an opening handshake and data transfer. The API enables Web pages to use the WebSocket protocol for two-way communication with the remote host.

JSR 356[4] defines a standard API for creating WebSocket applications in the Java EE 7 Platform. The JSR provides support for:

- Create WebSocket endpoint using annotations and interface
- Initiating and intercepting WebSocket events
- Creation and consumption of WebSocket text and binary messages
- Configuration and management of WebSocket sessions
- Integration with Java EE security model

This section will build a chat room for movie viewers.

1. Right-click on 'Source Packages' , select 'New', 'Java Class'. Give the class name as 'ChatServer', package as 'org.javaee7.movieplex7.chat', and click on 'Finish'.
2. Change the class such that it looks like:

```
@ServerEndpoint("/websocket")
public class ChatServer {
    private static final Set<Session> peers =
                Collections.synchronizedSet(new HashSet<Session>());

    @OnOpen
```

[1] http://tools.ietf.org/html/rfc6455
[2] http://tools.ietf.org/html/rfc6455
[3] http://www.w3.org/TR/websockets/
[4] http://jcp.org/en/jsr/detail?id=356

```
    public void onOpen(Session peer) {
        peers.add(peer);
    }

    @OnClose
    public void onClose(Session peer) {
        peers.remove(peer);
    }

    @OnMessage
    public void message(String message, Session client)
            throws IOException, EncodeException {
        for (Session peer : peers) {
            peer.getBasicRemote().sendText(message);
        }
    }
}
```

In this code:

a. `@ServerEndpoint` decorates the class to be a WebSocket endpoint. The value defines the URI where this endpoint is published.

b. `@OnOpen` and `@OnClose` decorate the methods that must be called when WebSocket session is opened or closed. The peer parameter defines the client requesting connection initiation and termination.

c. `@OnMessage` decorates the message that receives the incoming WebSocket message. The first parameter, message, is the payload of the message. The second parameter, `client`, defines the other end of the WebSocket connection. The method implementation transmits the received text message to all clients connected to this endpoint.
Resolve the imports by right-clicking in the editor and selecting ‘Fix Imports’ or (Command + Shift + I shortcut on Mac or Ctrl + Shift + I on Windows).

Make sure to pick `java.websocket.Session` for resolving imports. This is not the default option shown by NetBeans.

Figure 4.1.

Right-click again in the editor pane and select ‘Format’ to format your code.

3. In ‘Web Pages’, select ‘New’, ‘Folder’, give the folder name as ‘chat’ and click on ‘Finish’.

4. Right-click on the newly created folder, select ‘New’, ‘Other’, ‘Java Server Faces’, ‘Facelets Template Client’, give the File Name as ‘chatroom’. Click on ‘Browse’ next to ‘Template:’, expand ‘Web Pages’, ‘WEB-INF’, select ‘template.xhtml’, and click on ‘Select File’. Click on ‘Finish’.

Figure 4.2.

In this file, remove <ui:define> sections where name attribute value is ‘top’ and ‘left’. These sections are inherited from the template.

Replace `<ui:define>` section with ‘content’ name such that it looks like:

```
<ui:define name="content">
    <form action="">
        <table>
            <tr>
                <td>
                    Chat Log<br/>

  <textarea readonly="true" rows="6" cols="50" id="chatlog"></textarea>
                </td>
                <td>
                    Users<br/>

  <textarea readonly="true" rows="6" cols="20" id="users"></textarea>
                </td>
            </tr>
            <tr>
                <td colspan="2">

  <input id="textField" name="name" value="Duke" type="text"/>

  <input onclick="join();" value="Join" type="button"/>

  <input onclick="send_message();" value="Send" type="button"/><p/>

  <input onclick="disconnect();" value="Disconnect" type="button"/>
                </td>
            </tr>
        </table>
    </form>
    <div id="output"></div>
        <script language="javascript" type="text/javascript"
            src="${facesContext.externalContext.requestContextPath}/
chat/websocket.js"></script>
</ui:define>
```

The code builds an HTML form that has two textareas – one to display the chat log and the other to display the list of users currently logged. A single text box is used to take the user name or the chat message. Clicking on ‘Join’ button takes the value as user name and clicking on ‘Send’ takes the value as chat message.

JavaScript methods are invoked when these buttons are clicked and these are explained in the next section. The chat messages are sent and received as WebSocket payloads. There is an explicit button to disconnect the WebSocket

connection. `output` div is the placeholder for status messages. The WebSocket initialization occurs in 'websocket.js' included at the bottom of the fragment.

5. Right-click on 'chat' in 'Web Pages', select 'New', 'Web' categories, 'JavaScript File' file type. Click on 'Next'.
   Give the name as 'websocket' and click on 'Finish'.

6. Edit the contents of 'websocket.js' such that it looks like:

```
var wsUri = 'ws://' + document.location.host
            + document.location.pathname.substr(0,
              document.location.pathname.indexOf("/faces")) +
              '/websocket';
console.log(wsUri);

var websocket = new WebSocket(wsUri);
var textField = document.getElementById("textField");
var users = document.getElementById("users");
var chatlog = document.getElementById("chatlog");
var username;

websocket.onopen = function(evt) { onOpen(evt); };
websocket.onmessage = function(evt) { onMessage(evt); };
websocket.onerror = function(evt) { onError(evt); };
websocket.onclose = function(evt) { onClose(evt); };

var output = document.getElementById("output");

function join() {
    username = textField.value;
    websocket.send(username + " joined");
}

function send_message() {
    websocket.send(username + ": " + textField.value);
}

function onOpen() {
    writeToScreen("CONNECTED");
}

function onClose() {
    writeToScreen("DISCONNECTED");
}

function onMessage(evt) {
```

```
    writeToScreen("RECEIVED: " + evt.data);
    if (evt.data.indexOf("joined") !== -1) {
        users.innerHTML += evt.data.substring(0, evt.data.indexOf("
 joined")) + "\n";
    } else {
        chatlog.innerHTML += evt.data + "\n";
    }
}

function onError(evt) {
    writeToScreen('<span style="color: red;">ERROR:</span> ' +
 evt.data);
}

function disconnect() {
    websocket.close();
}

function writeToScreen(message) {
    var pre = document.createElement("p");
    pre.style.wordWrap = "break-word";
    pre.innerHTML = message;
    output.appendChild(pre);
}
```

The WebSocket endpoint URI is calculated by using standard JavaScript variables and appending the URI specified in the `ChatServer` class. WebSocket is initialized by calling new `WebSocket(...)`. Event handlers are registered for lifecycle events using `onXXX` messages. The listeners registered in this script are explained in the table.

Listeners	Called When
`onOpen(evt)`	WebSocket connection is initiated
`onMessage(evt)`	WebSocket message is received
`onError(evt)`	Error occurs during the communication
`onClose(evt)`	WebSocket connection is terminated

Any relevant data is passed along as parameter to the function. Each method prints the status on the browser using `writeToScreen` utility method. The join method sends a message to the endpoint that a particular user has joined. The endpoint then broadcasts the message to all the listening clients. The `send_message` method appends the logged in user name and the value of the text field and

broadcasts to all the clients similarly. The `onMessage` method updates the list of logged in users as well.

7. Edit 'WEB-INF/template.xhtml' and change:

```
<h:outputLink value="item2.xhtml">Item 2</h:outputLink>
```

to

```
<h:outputLink
    value="${facesContext.externalContext.requestContextPath}/faces/
chat/chatroom.xhtml">
    Chat Room
</h:outputLink>
```

The `outputLink` tag renders an HTML anchor tag with an `href` attribute. `${facesContext.externalContext.requestContextPath}` provides the request URI that identifies the web application context for this request. This allows the links in the left navigation bar to be fully-qualified URLs.

8. Run the project by right clicking on the project and selecting 'Run'. The browser shows localhost:8080/movieplex7[5].

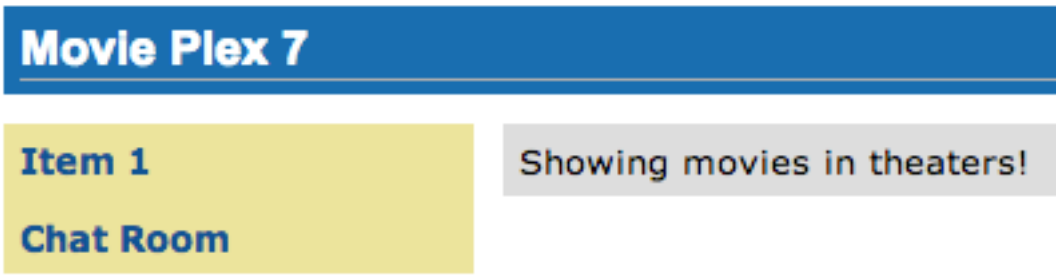


Figure 4.3.

Click on 'Chat Room' to see the output.

The 'CONNECTED' status message is shown and indicates that the WebSocket connection with the endpoint is established.

[5] http://localhost:8080/movieplex7

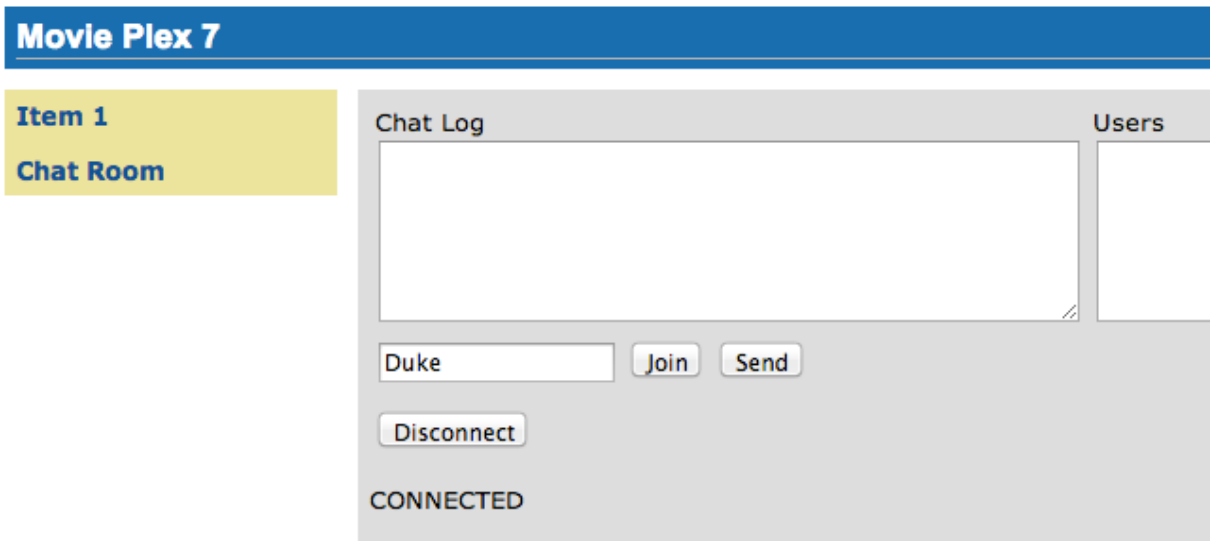


Figure 4.4.

Please make sure your browser supports WebSocket in order for this page to show up successfully. Chrome 14.0+, Firefox 11.0+, Safari 6.0+, and IE 10.0+ are the browsers that support WebSocket. A complete list of supported browsers is available at caniuse.com/websockets[6].

Open the URI localhost:8080/movieplex7[7] in another browser window. Enter 'Duke' in the text box in the first browser and click 'Join'.

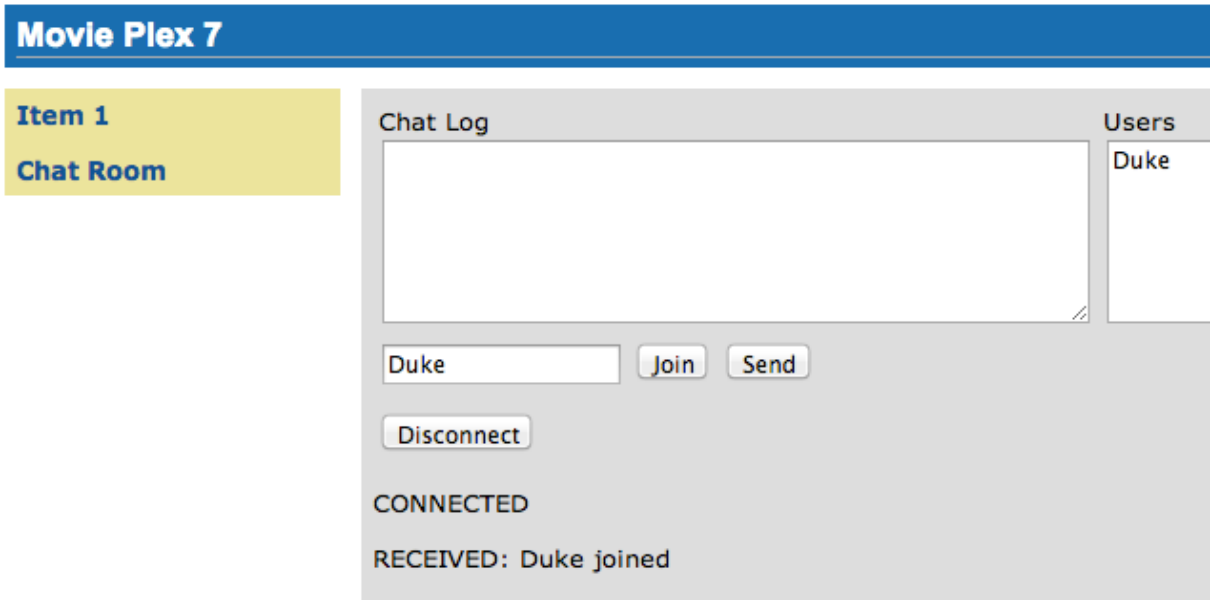


Figure 4.5.

[6] http://caniuse.com/websockets
[7] http://localhost:8080/movieplex7

Notice that the user list and the status message in both the browsers gets updated. Enter ‘James’ in the text box of the second browser and click on ‘Join’. Once again the user list and the status message in both the browsers is updated. Now you can type any messages in any of the browser and click on ‘Send’ to send the message.

The output from two different browsers after the initial greeting looks like as shown.

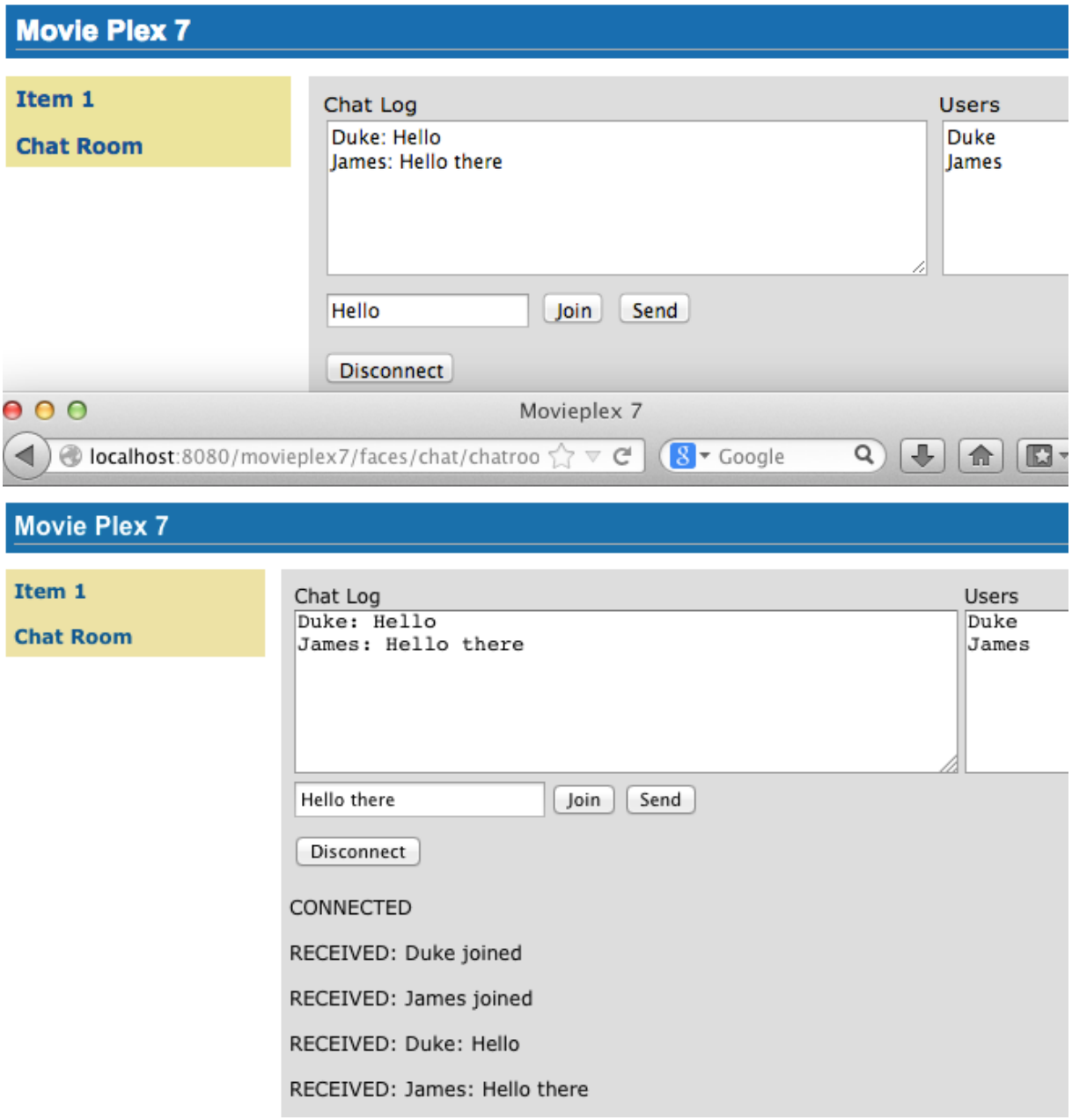


Figure 4.6.

Here it shows output from Chrome on the top and Firefox on the bottom.

Chrome Developer Tools or Firebug in Firefox can be used to monitor WebSocket traffic.

Chapter 5. Ticket Sales (Batch Applications for the Java Platform)

Purpose: Read the total sales for each show and populate the database. In doing so several new features of Java API for Batch Processing 1.0 will be introduced and demonstrated by using them in the application.

Estimated Time: 30-45 mins

Batch Processing is execution of series of 'jobs' that is suitable for non-interactive, bulk-oriented and long-running tasks. Batch Applications for the Java Platform (JSR 352) will define a programming model for batch applications and a runtime for scheduling and executing jobs.

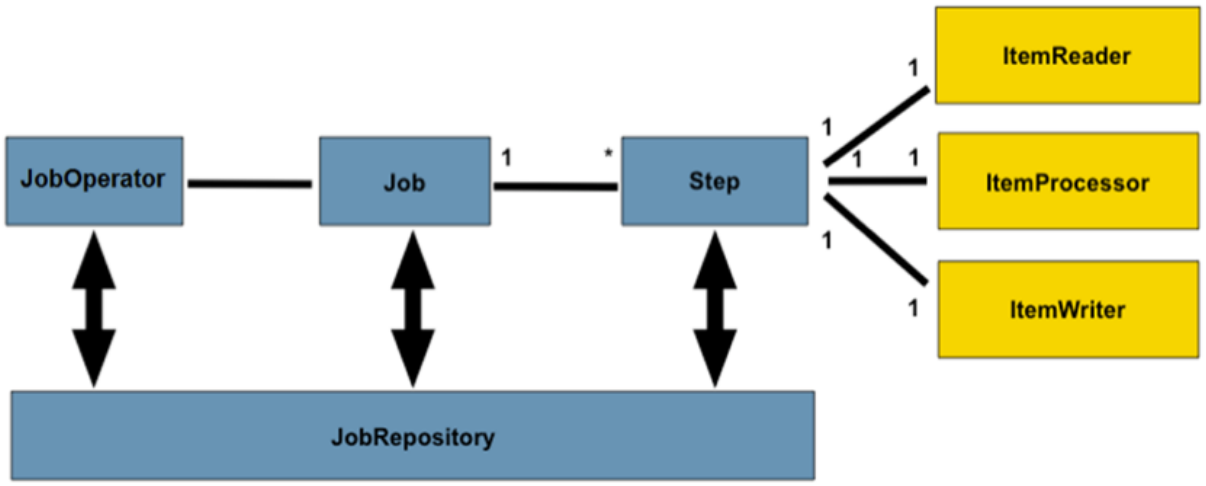


Figure 5.1.

The core concepts of Batch Processing are:

- A **Job** is an instance that encapsulates an entire batch process. A job is typically put together using a Job Specification Language and consists of multiple steps. The Job Specification Language for JSR 352 is implemented with XML and is referred as 'Job XML'.
- A **Step** is a domain object that encapsulates an independent, sequential phase of a job. A step contains all of the information necessary to define and control the actual batch processing.

- **JobOperator** provides an interface to manage all aspects of job processing, including operational commands, such as start, restart, and stop, as well as job repository commands, such as retrieval of job and step executions.
- **JobRepository** holds information about jobs current running and jobs that run in the past. JobOperator provides access to this repository.
- Reader-Processor-Writer pattern is the primary pattern and is called as **Chunk-oriented Processing**. In this, **ItemReader** reads one item at a time, **ItemProcessor** processes the item based upon the business logic, such as calculate account balance and hands it to **ItemWriter** for aggregation. Once the *chunk* numbers of items are aggregated, they are written out, and the transaction is committed.

This section will read the cumulative sales for each show from a CSV file and populate them in a database.

1. Right-click on Source Packages, select ‘New’, ‘Java Package’, specify the value as ‘org.javaee7.movieplex7.batch’, and click on ‘Finish’.
2. Right-click on newly created package, select ‘New’, ‘Java Class’, specify the name as ‘SalesReader’. Make this class extend from ‘AbstractItemReader’ by changing the class definition and add:

```
extends AbstractItemReader
```

`AbstractItemReader` is an abstract class that implements `ItemReader` interface. The `ItemReader` interface defines methods that read a stream of items for chunk processing. This reader implementation returns a String item type as indicated in the class definition.

Add `@Named` as a class-level annotations and it allows the bean to be injected in Job XML. Add `@Dependent` as another class-level annotation to mark this bean as a bean defining annotation so that this bean is available for injection.

Resolve the imports.

3. Override `open()` method to initialize the reader by adding the following code:

```
private BufferedReader reader;

public void open(Serializable checkpoint) throws Exception {
    reader = new BufferedReader(
        new InputStreamReader(
        Thread.currentThread()
```

```
                .getContextClassLoader()
                .getResourceAsStream("META-INF/sales.csv")));
    }
```

This method initializes a BufferedReader from 'META-INF/sales.csv' that is bundled with the application.

Sampling of the first few lines from 'sales.csv' is shown below:

```
1,500.00
2,660.00
3,80.00
4,470.00
5,1100.x0
```

Each line has a show identifier comma separated by the total sales for that show. Note that the last line (5th record in the sample) has an intentional typo. In addition, 17th record also has an additional typo. The lab will use these lines to demonstrate how to handle parsing errors.

4. Override the following method from the abstract class:

```
@Override
public String readItem() {
    String string = null;
    try {
        string = reader.readLine();
    } catch (IOException ex) {
        ex.printStackTrace();
    }
    return string;
}
```

The `readItem` method returns the next item from the stream. It returns `null` to indicate end of stream. Note end of stream indicates end of chunk, so the current chunk will be committed and the step will end.

Resolve the imports.

5. Right-click on 'org.javaee7.movieplex7.batch' package, select 'New', 'Java Class', specify the name as 'SalesProcessor'. Change the class definition and add:

```
implements ItemProcessor
```

`ItemProcessor` is an interface that defines a method that is used to operate on an input item and produce an output item. This processor accepts a String input item from the reader, `SalesReader` in our case, and returns a `Sales` instance to the writer (coming shortly). `Sales` is the pre-packaged JPA entity with the application starter source code.

Add `@Named` and `@Dependent` as class-level annotations so that it allows the bean to be injected in Job XML.

Resolve the imports.

6. Add implementation of the abstract method from the interface as:

```
@Override
public Sales processItem(Object s) {
    Sales sales = new Sales();
    StringTokenizer tokens = new StringTokenizer((String)s, ",");
    sales.setId(Integer.parseInt(tokens.nextToken()));
    sales.setAmount(Float.parseFloat(tokens.nextToken()));

    return sales;
}
```

This method takes a `String` parameter coming from the `SalesReader`, parses the value, populates them in the `Sales` instance, and returns it. This is then aggregated with the writer.

The method can return null indicating that the item should not be aggregated. For example, the parsing errors can be handled within the method and return null if the values are not correct. However this method is implemented where any parsing errors are thrown as exception. Job XML can be instructed to skip these exceptions and thus that particular record is skipped from aggregation as well (shown later).

Resolve the imports.

7. Right-click on `org.javaee7.movieplex7.batch` package, select ‘New’, ‘Java Class’, specify the name as ‘SalesWriter’. Change the class definition and add:

```
extends AbstractItemWriter
```

`AbstractItemWriter` is an abstract class that implements `ItemWriter` interface. The ItemWriter interface defines methods that write to a stream of items for chunk processing. This writer writes a list of `Sales` items.

Add `@Named` and `@Dependent` as class-level annotations so that it allows the bean to be injected in Job XML.

Resolve the imports.

8. Inject `EntityManager` as:

```
@PersistenceContext EntityManager em;
```

Override `writeItems` method from the abstract class by adding the following code:

```
@Override
@Transactional
public void writeItems(List list) {
    for (Sales s : (List<Sales>)list) {
        em.persist(s);
    }
}
```

Batch runtime aggregates the list of `Sales` instances returned from the `SalesProessor` and makes it available as List in this method. This method iterates over the list and persist each item in the database.

The method also specifies `@Transactional` as a method level annotation. This is a new annotation introduced by JTA 1.2 that provides the ability to control transaction boundaries on CDI managed beans. This provides the semantics of EJB transaction attributes in CDI beans without dependencies such as RMI. This support is implemented via an implementation of a CDI interceptor that conducts the necessary suspending, resuming, etc.

In this case, a transaction is automatically started before the method is called, committed if no checked exceptions are thrown, and rolled back if runtime exceptions are thrown. This behavior can be overridden using `rollbackOn` and `dontRollbackOn` attributes of the annotation.

> Each chunk is processed within a container-managed transaction already. There is really no need for `@Transactional` on `writeItems` method but shows a usage for the annotation.

Resolve the imports.

9. Create Job XML that defines the job, step, and chunk.
   In 'Files' tab, expand the project → 'src' → 'main' → 'resources', right-click on 'META-INF', select 'New', 'Folder', specify the name as 'batch-jobs', and click on 'Finish'.

   Right-click on the newly created folder, select 'New', 'Other', select 'XML', 'XML Document', click on 'Next >', give the name as 'eod-sales', click on 'Next', take the default, and click on 'Finish'.

   Replace contents of the file with the following:

```
<job id="endOfDaySales"
    xmlns="http://xmlns.jcp.org/xml/ns/javaee[http://xmlns.jcp.org/xml/
ns/javaee]"
    version="1.0">
    <step id="populateSales">
        <chunk item-count="3" skip-limit="5">
            <reader ref="salesReader"/>
            <processor ref="salesProcessor"/>
            <writer ref="salesWriter"/>
            <skippable-exception-classes>
                <include class="java.lang.NumberFormatException"/>
            </skippable-exception-classes>
        </chunk>
    </step>
</job>
```

   This code shows that the job has one step of chunk type. The `<reader>`, `<processor>`, and `<writer>` elements define the CDI bean name of the implementations of `ItemReader`, `ItemProcessor`, and `ItemWriter` interfaces. The `item-count` attribute defines that 3 items are read/processed/aggregated and then given to the writer. The entire reader/processor/writer cycle is executed within a transaction.

   The `<skippable-exception-classes>` element specifies a set of exceptions to be skipped by chunk processing.

   CSV file used for this lab has intentionally introduced couple of typos that would generate `NumberFormatException`. Specifying this element allows skipping the exception, ignore that particular element, and continue processing. If this element

is not specified then the batch processing will halt. The `skip-limit` attribute specifies the number of exceptions a step will skip.

10 Lets invoke the batch job.

In 'Projects' tab, right-click on 'org.javaee7.movieplex7.batch' package, select 'New', 'Java Class'. Enter the name as 'SalesBean' and click on 'Finish' button.

Add the following code to the bean:

```
public void runJob() {
    try {
        JobOperator jo = BatchRuntime.getJobOperator();
        long jobId = jo.start("eod-sales", new Properties());
        System.out.println("Started job: with id: " + jobId);
    } catch (JobStartException ex) {
        ex.printStackTrace();
    }
}
```

This method uses `BatchRuntime` to get an instance of `JobOperator`, which is then used to start the job. `JobOperator` is the interface for operating on batch jobs. It can be used to start, stop, and restart jobs. It can additionally inspect job history, to discover what jobs are currently running and what jobs have previously run.

Add `@Named` and `@RequestScoped` as class-level annotations. This allows the bean to be injectable in an EL expression.

Resolve the imports.

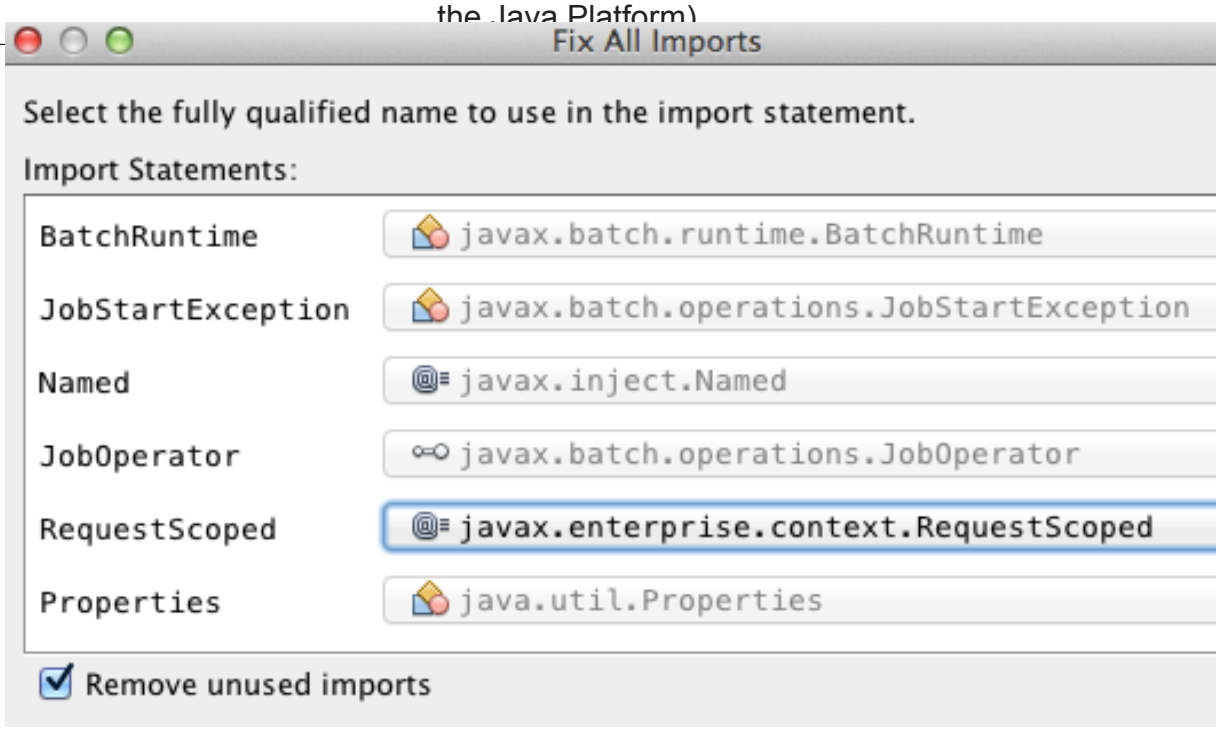


Figure 5.2.

11. Inject `EntityManagerFactory` in the class as:

```
@PersistenceUnit EntityManagerFactory emf;
```

and add the following method:

```
public List<Sales> getSalesData() {
    return emf.
        createEntityManager().
        createNamedQuery("Sales.findAll", Sales.class).
        getResultList();
}
```

This method uses a pre-defined `@NamedQuery` to query the database and return all the rows from the table.

Resolve the imports.

12. Right-click on ‘Web Pages’, select ‘New’, ‘Folder’, specify the name as ‘batch’, and click on ‘Finish’.
Right-click on the newly created folder, select ‘New’, ‘Other’, ‘JavaServer Faces’, ‘Facelets Template Client’, and click on ‘Next >’.

Give the File Name as ‘sales’. Click on ‘Browse’ next to ‘Template:’, expand ‘Web Pages’, ‘WEB-INF’, select ‘template.xhtml’, and click on ‘Select File’. Click on ‘Finish’.

In this file, remove `<ui:define>` sections where name attribute value is ‘top’ and ‘left’. These sections are inherited from the template.

Replace `<ui:define>` section with ‘content’ name such that it looks like:

```
<ui:define name="content">
    <h1>Movie Sales</h1>
    <h:form>
        <h:dataTable value="#{salesBean.salesData}" var="s" border="1">
            <h:column>
                <f:facet name="header">
                    <h:outputText value="Show ID" />
                </f:facet>
                #{s.id}
            </h:column>
            <h:column>
                <f:facet name="header">
                    <h:outputText value="Sales" />
                </f:facet>
                #{s.amount}
            </h:column>
        </h:dataTable>
        <h:commandButton
            value="Run Job"
            action="sales"
            actionListener="#{salesBean.runJob()}"/>
        <h:commandButton
            value="Refresh"
            action="sales" />
    </h:form>
</ui:define>
```

This code displays the show identifier and sales from that show in a table by invoking `SalesBean.getSalesData()`. First command button allows invoking the job that processes the CSV file and populates the database. The second command button refreshes the page.

Right-click on the yellow bulb to fix namespace prefix/URI mapping for `h:`. This needs to be repeated for `f:` prefix.

13. Add the following code in `template.xhtml` along with other <outputLink>s:

```
<p/><h:outputLink
    value="${facesContext.externalContext.requestContextPath}/faces/
batch/sales.xhtml">
    Sales
    </h:outputLink>
```

14. Run the project to see the output as shown.

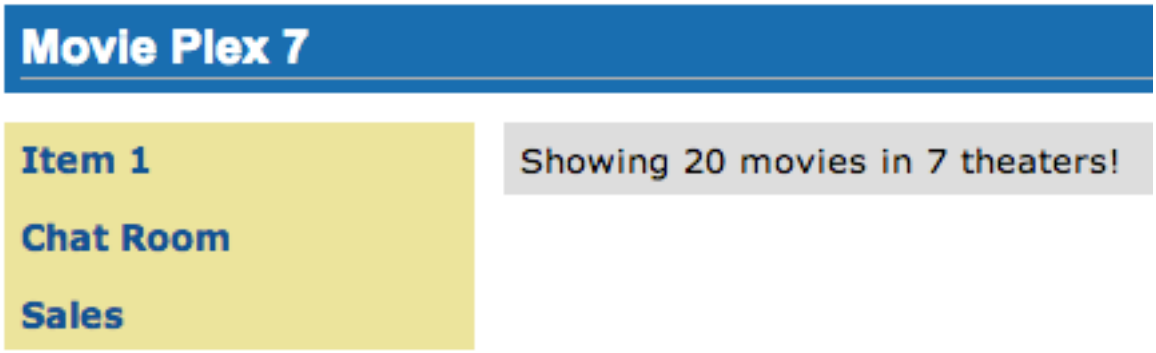


Figure 5.3.

Notice, a new 'Sales' entry is displayed in the left navigation bar.

15. Click on 'Sales' to see the output as shown.

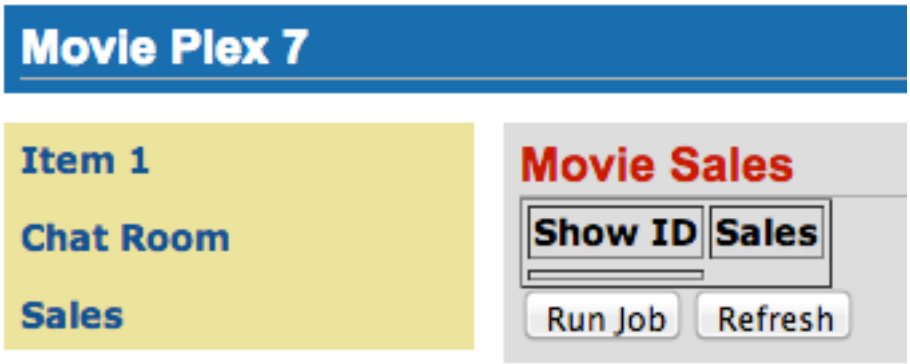


Figure 5.4.

The empty table indicates that there is no sales data in the database.

16. Click on 'Run Job' button to initiate data processing of CSV file. Look for 'Waiting for localhost' in the browser status bar, wait for a couple of seconds for the processing to finish, and then click on 'Refresh' button to see the updated output as shown.

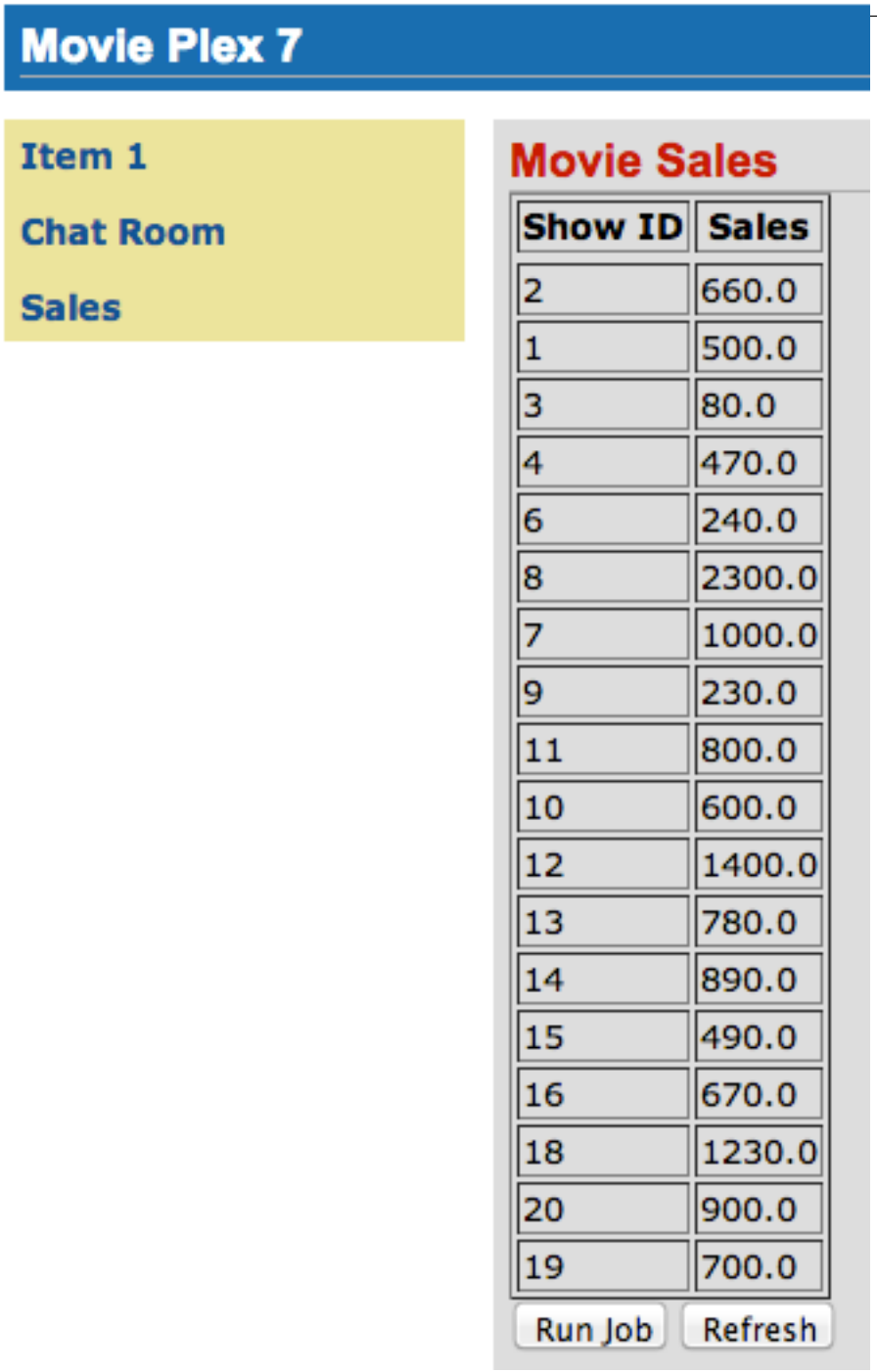


Figure 5.5.

Now the table is populated with the sales data.

Note that record 5 is missing from the table, as this records did not have correct numeric entries for the sales total. The Job XML for the application explicitly mentioned to skip such errors.

Chapter 6. View and Delete Movie (Java API for RESTful Web Services)

Purpose: View, and delete a movie. In doing so several new features of JAX-RS 2 will be introduced and demonstrated by using them in the application.

Estimated Time: 30-45 mins

JAX-RS 2 defines a standard API to create, publish, and invoke a REST endpoint. JAX-RS 2 adds several new features to the API:

- Client API that can be used to access Web resources and provides integration with JAX-RS Providers. Without this API, the users need to use a low-level HttpUrlConnection to access the REST endpoint.
- Asynchronous processing capabilities in Client and Server that enables more scalable applications.
- Message Filters and Entity Interceptors as well-defined extension points to extend the capabilities of an implementation.
- Validation constraints can be specified to validate the parameters and return type.

This section will provide the ability to view all the movies, details of a selected movie, and delete an existing movie using the JAX-RS Client API.

1. Right-click on ‘Source Packages’, select ‘New’, ‘Java Class’. Give the class name as ‘MovieClientBean’, package as ‘org.javaee7.movieplex7.client’, and click on ‘Finish’.
   This bean will be used to invoke the REST endpoint.
2. Add `@Named` and `@RequestScoped` class-level annotations. This allows the class to be injected in an EL expression and also defines the bean to be automatically activated and passivated with the request.
   Resolve the imports.

Make sure to pick `javax.enterprise.context.RequestScoped` class.

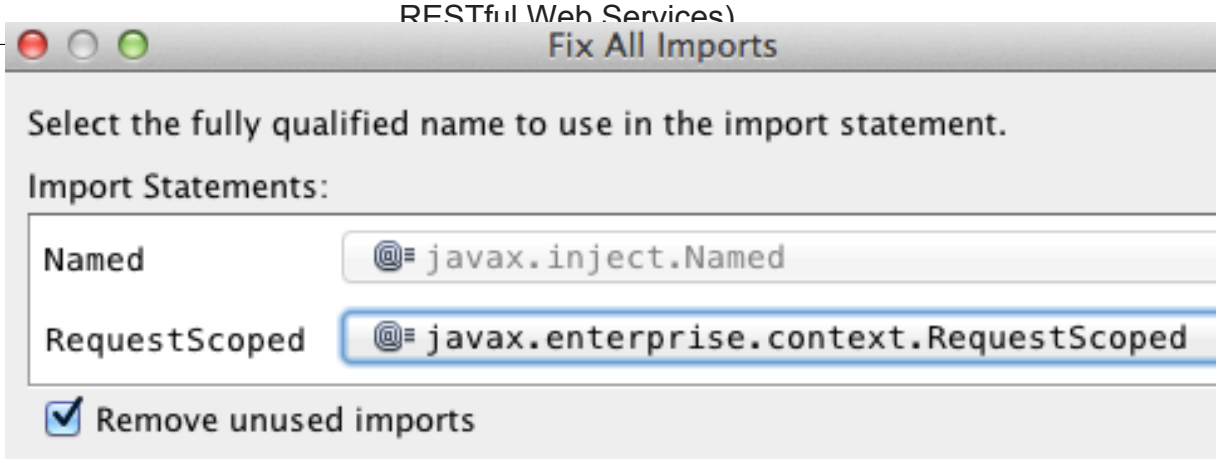


Figure 6.1.

3. Add the following code to the class:

```
Client client;
WebTarget target;

@Inject HttpServletRequest httpServletRequest;

@PostConstruct
public void init() {
    client = ClientBuilder.newClient();
    target = client
                .target("http://" +
                httpServletRequest.getLocalName() +
                ":" +
                httpServletRequest.getLocalPort() +
                "/" +
                httpServletRequest.getContextPath() +
                "/webresources/movie/");

}

@PreDestroy
public void destroy() {
    client.close();
}
```

`ClientBuilder` is the main entry point to the Client API. It uses a fluent builder API to invoke REST endpoints. A new `Client` instance is created using the default client builder implementation provided by the JAX-RS implementation provider. Client are heavy-weight objects that manage the client-side communication

infrastructure. It is highly recommended to create only required number of instances of Client and close it appropriately.

In this case, `Client` instance is created and destroyed in the lifecycle callback methods. The endpoint URI is set on this instance by calling the target method. Note that the endpoint address is dynamically created by injecting an instance of `HttpServletRequest`. This is a new feature added in CDI 1.1

4. Add the following method to the class:

```
public Movie[] getMovies() {
    return target
        .request()
        .get(Movie[].class);
}
```

A request is prepared by calling the request method. HTTP GET method is invoked by calling get method. The response type is specified in the last method call and so return value is of the type `Movie[]`.

5. Right-click on 'Web Pages', select 'New', 'Folder', specify the name as 'client', and click on 'Finish'.
Right-click on the newly created folder, select 'New', 'Other', 'JavaServer Faces', 'Facelets Template Client', and click on 'Next >'.

   Give the File Name as 'movies'. Click on 'Browse' next to 'Template:', expand 'Web Pages', 'WEB-INF', select 'template.xhtml', and click on 'Select File'. Click on 'Finish'.

6. In this file, remove `<ui:define>` sections where name attribute value is 'top' and 'left'. These sections are inherited from the template.
Replace `<ui:define>` section with 'content' name such that it looks like:

```
<ui:define name="content">
    <h:form prependId="false">

  <h:selectOneRadio value="#{movieBackingBean.movieId}" layout="pageDirection">
            <c:forEach items="#{movieClientBean.movies}" var="m">

  <f:selectItem itemValue="#{m.id}" itemLabel="#{m.name}"/>
            </c:forEach>
        </h:selectOneRadio>
        <h:commandButton value="Details" action="movie" />
    </h:form>
```

```
</ui:define>
```

This code fragment invokes `getMovies` method from `MovieClientBean`, iterates over the response in a for loop, and display the name of each movie with a radio button. The selected radio button value is bound to the EL expression `#{movieBackingBean.movieId}`.

The code also has a button with 'Details' label and looks for 'movie.xhtml' in the same directory. We will create this file later.

Click on the yellow bulb in the left bar to resolve the namespace prefix-to-URI resolution. This needs to be completed for `h:`, `c:`, and `f:` prefixes.

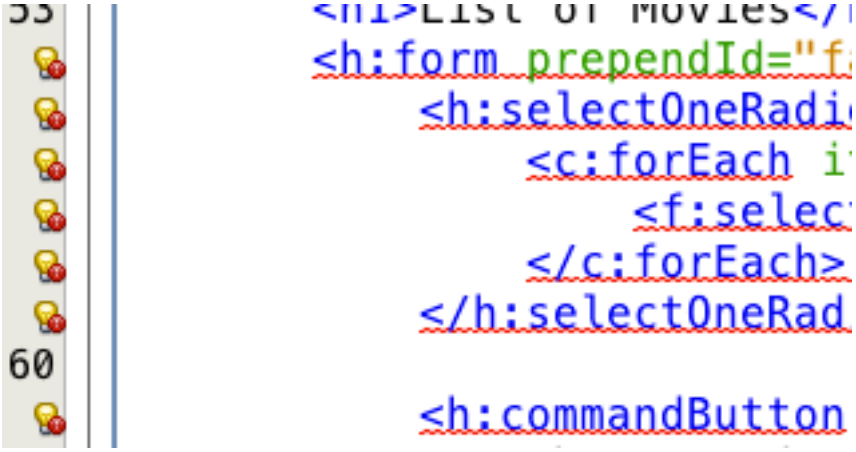


Figure 6.2.

7. Right-click on 'org.javaee7.movieplex7.client' package, select 'New', 'Java Class', specify the value as 'MovieBackingBean' and click on 'Finish'.
   Add the following field:

```
int movieId;
```

Add getters/setters by right-clicking on the editor pane and selecting 'Insert Code' (Ctrl + I shortcut on Mac). Select the field and click on 'Generate'.

Add `@Named` and `@SessionScoped` class-level annotations and implements `Serializable`.

Resolve the imports.

Make sure to import `javax.enterprise.context.SessionScoped`.

8. In ‘template.xhtml’, add the following code along with other <outputLink>s:

```
<p/><h:outputLink
        value="${facesContext.externalContext.requestContextPath}/
faces/client/movies.xhtml">
        Movies
    </h:outputLink>
```

Running the project (Fn + F6 shortcut on Mac) and clicking on ‘Movies’ in the left navigation bar shows the output as shown.

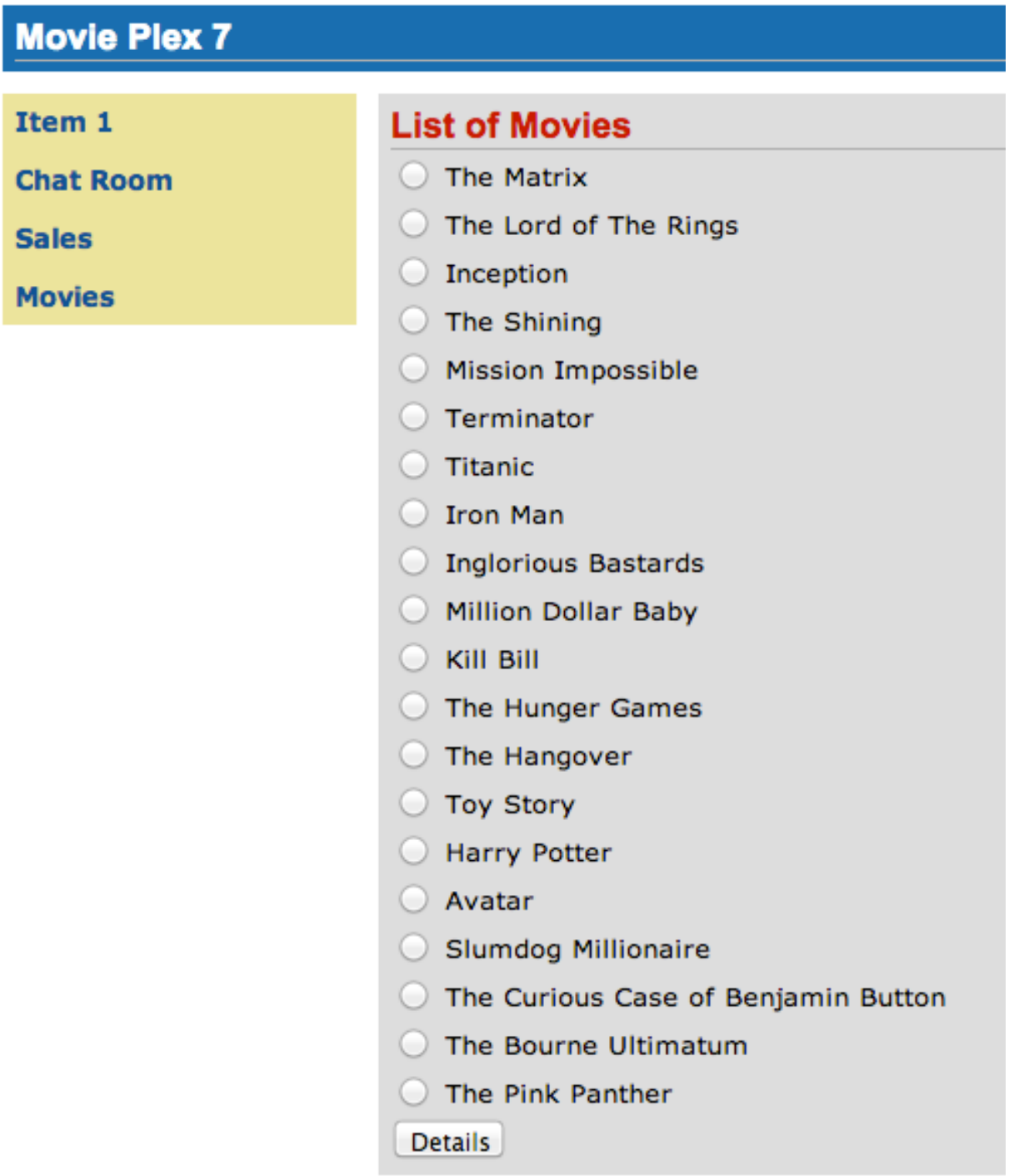


Figure 6.3.

The list of all the movies with a radio button next to them is displayed.

9. In `MovieClientBean`, inject `MovieBackingBean` to read the value of selected movie from the page. Add the following code:

```
@Inject
```

```
MovieBackingBean bean;
```

10. In `MovieClientBean`, add the following method:

```
public Movie getMovie() {
    Movie m = target
        .path("{movie}")
        .resolveTemplate("movie", bean.getMovieId())
        .request()
        .get(Movie.class);
    return m;
}
```

This code reuses the `Client` and `WebTarget` instances created in `@PostConstruct`. It also adds a variable part to the URI of the REST endpoint, defined using `{movie}`, and binds it to a concrete value using `resolveTemplate` method. The return type is specified as a parameter to the get method.

11. Right-click on 'client' folder, select 'New', 'Facelets Template Client', give the File Name as 'movie'. Click on 'Browse' next to 'Template:', expand 'Web Pages', 'WEB-INF', select 'template.xhtml', and click on 'Select File'. Click on 'Finish'.

12. In this file, remove `<ui:define>` sections where name attribute value is 'top' and 'left'. These sections are inherited from the template.
Replace `<ui:define>` with 'content' name such that it looks like:

```
<ui:define name="content">
    <h1>Movie Details</h1>
    <h:form>
        <table cellpadding="5" cellspacing="5">
            <tr>
                <th align="left">Movie Id:</th>
                <td>#{movieClientBean.movie.id}</td>
            </tr>
            <tr>
                <th align="left">Movie Name:</th>
                <td>#{movieClientBean.movie.name}</td>
            </tr>
            <tr>
                <th align="left">Movie Actors:</th>
                <td>#{movieClientBean.movie.actors}</td>
            </tr>
        </table>
        <h:commandButton value="Back" action="movies" />
```

```
    </h:form>
</ui:define>
```

Click on the yellow-bulb to resolve the namespace prefix-URI mapping for `h:`.

The output values are displayed by calling the `getMovie` method and using the `id`, `name`, and `actors` property values.

13 Run the project, select 'Movies' in the left navigation bar, select a radio button next to any movie, and click on details to see the output as shown.

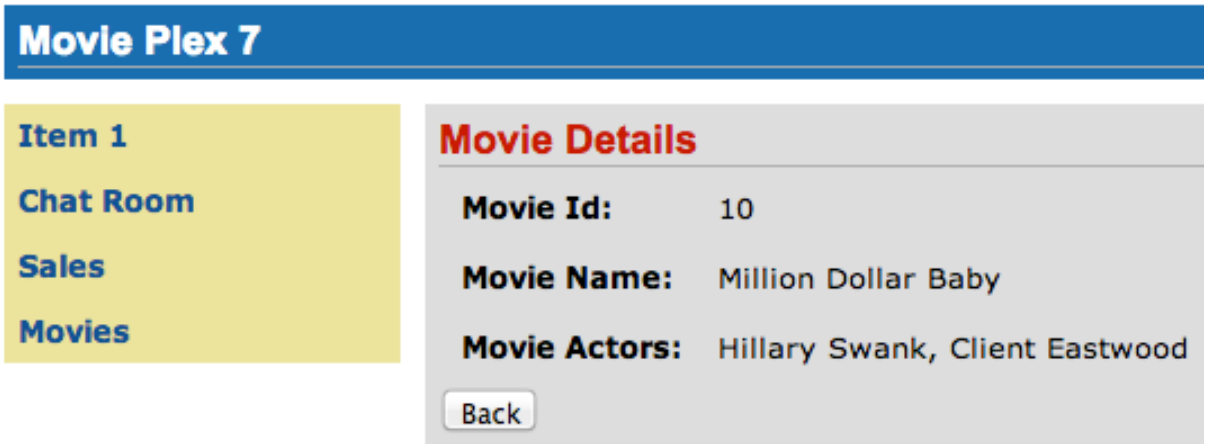


Figure 6.4.

Click on the 'Back' button to select another movie.

14 Add the ability to delete a movie. In 'movies.xhtml', add the following code with other <commandButton>.

```
<h:commandButton
    value="Delete"
    action="movies"
    actionListener="#{movieClientBean.deleteMovie()}"/>
```

This button displays a label 'Delete', invokes the method deleteMovie from 'MovieClientBean', and then renders 'movies.xhtml'.

15 Add the following code to 'MovieClientBean':

```
public void deleteMovie() {
    target
        .path("{movieId}")
        .resolveTemplate("movieId", bean.getMovieId())
        .request()
        .delete();
```

```
}
```

This code again reuses the `Client` and `WebTarget` instances created in `@PostConstruct`. It also adds a variable part to the URI of the REST endpoint, defined using `{movieId}`, and binds it to a concrete value using `resolveTemplate` method. The URI of the resource to be deleted is prepared and then delete method is called to delete the resource.

Make sure to resolve the imports.

Running the project shows the output shown.

Figure 6.5.

Select a movie and click on Delete button. This deletes the movie from the database and refreshes list on the page. Note that a redeploy of the project will delete all the movies anyway and add them all back.

Chapter 7. Add Movie (Java API for JSON Processing)

Purpose: Add a new movie. In doing so several new features of the Java API for JSON Processing 1.0 will be introduced and demonstrated by using them in the application.

Estimated Time: 30-45 mins

Java API for JSON Processing provides a standard API to parse and generate JSON so that the applications can rely upon a portable API. This API will provide:

- Produce/Consume JSON in a streaming fashion (similar to StAX API for XML)
- Build a Java Object Model for JSON (similar to DOM API for XML)

This section will define a JAX-RS Entity Providers that will allow reading and writing JSON for a Movie POJO. The JAX-RS Client API will request this JSON representation.

JAX-RS Entity Providers supply mapping services between on-the-wire representations and their associated Java types. Several standard Java types such as `String`, `byte[]`, `javax.xml.bind.JAXBElement`, `java.io.InputStream`, `java.io.File`, and others have a pre-defined mapping and is required by the specification. Applications may provide their own mapping to custom types using `MessageBodyReader` and `MessageBodyWriter` interfaces.

This section will provide the ability to add a new movie to the application. Typically, this functionality will be available after proper authentication and authorization.

1. Right-click on Source Packages, select 'New', 'Java Class', specify the name as 'MovieReader', package as 'org.javaee7.movieplex7.json' and click on 'Finish'. Add the following class-level annotations:
2. Right-click on newly created package, select 'New', 'Java Class', specify the name as 'MovieReader', and click on 'Finish'. Add the following class-level annotations:

```
@Provider
@Consumes(MediaType.APPLICATION_JSON)
```

`@Provider` allows this implementation to be discovered by the JAX-RS runtime during the provider scanning phase. `@Consumes` indicates that this implementation will consume a JSON representation of the resource.

Make sure to resolve imports from the appropriate package as shown.

Figure 7.1.

3. Make the class implements `MessageBodyReader<Movie>`.

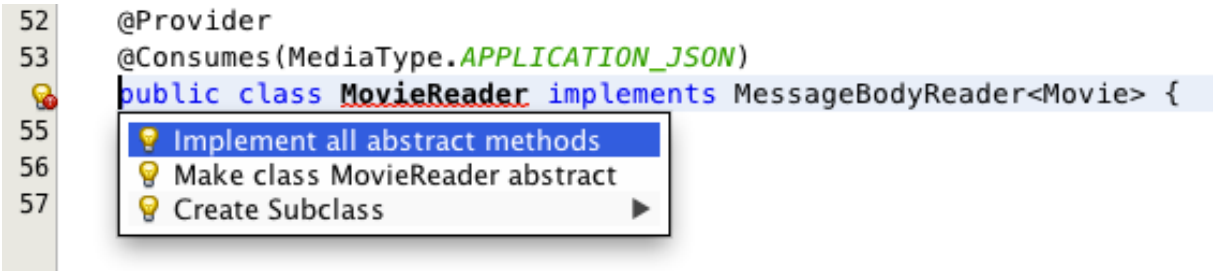


Figure 7.2.

Click on the hint (shown as yellow bulb) on the class definition and select 'Implement all abstract methods'.

4. Change implementation of the `isReadable` method as:

```
return Movie.class.isAssignableFrom(type);
```

This method ascertains if the `MessageBodyReader` can produce an instance of a particular type.

5. Replace the `readFrom` method with:

```
@Override
public Movie readFrom(
    Class<Movie> type,
```

```
        Type type1,
        Annotation[] antns,
        MediaType mt,
        MultivaluedMap<String, String> mm,
        InputStream in)
          throws IOException, WebApplicationException {

        Movie movie = new Movie();
        JsonParser parser = Json.createParser(in);
        while (parser.hasNext()) {
            switch (parser.next()) {
                case KEY_NAME:
                    String key = parser.getString();
                    parser.next();
                    switch (key) {
                        case "id":
                            movie.setId(parser.getInt());
                            break;
                        case "name":
                            movie.setName(parser.getString());
                            break;
                        case "actors":
                            movie.setActors(parser.getString());
                            break;
                        default:
                            break;
                    }
                    break;
                default:
                    break;
            }
        }
        return movie;
    }
```

This code reads a type from the input stream in. `JsonParser`, a streaming parser, is created from the input stream. Key values are read from the parser and a `Movie` instance is populated and returned.

Resolve the imports.

6. Right-click on 'org.javaee7.movieplex7.json' package, select 'New', 'Java Class', specify the name as 'MovieWriter', and click on 'Finish'. Add the following class-level annotations:

```
@Provider
@Produces(MediaType.APPLICATION_JSON)
```

`@Provider` allows this implementation to be discovered by the JAX-RS runtime during the provider scanning phase. `@Produces` indicates that this implementation will produce a JSON representation of the resource.

Resolve the imports as shown.

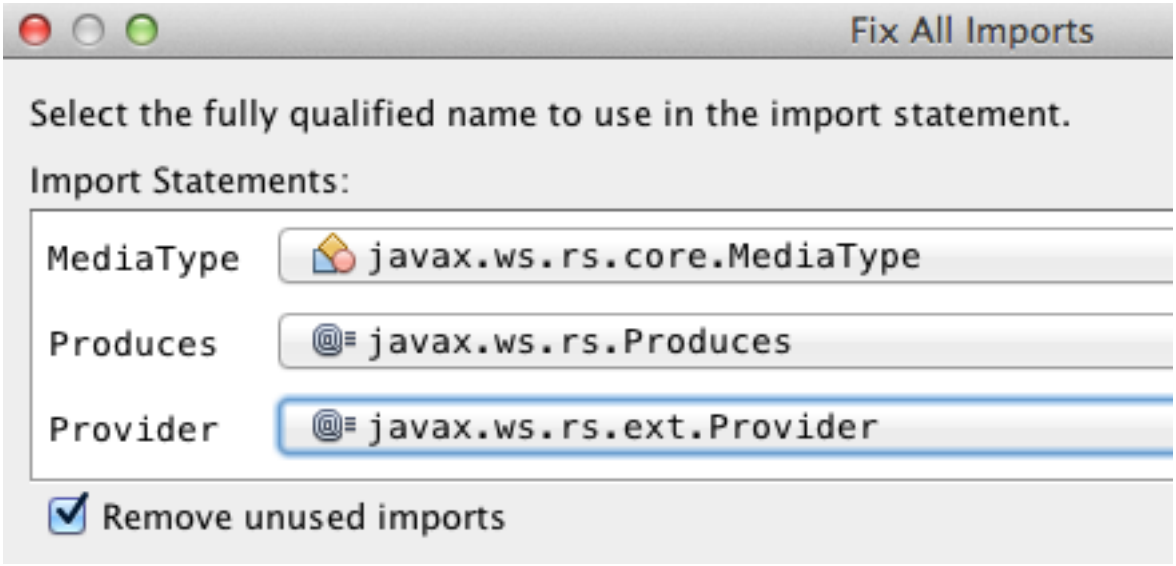


Figure 7.3.

7. Make this class implement `MessageBodyWriter` interface by adding the following code:

```
implements MessageBodyWriter<Movie>
```

Resolve the imports.

The IDE provide a hint to implement abstract methods as:

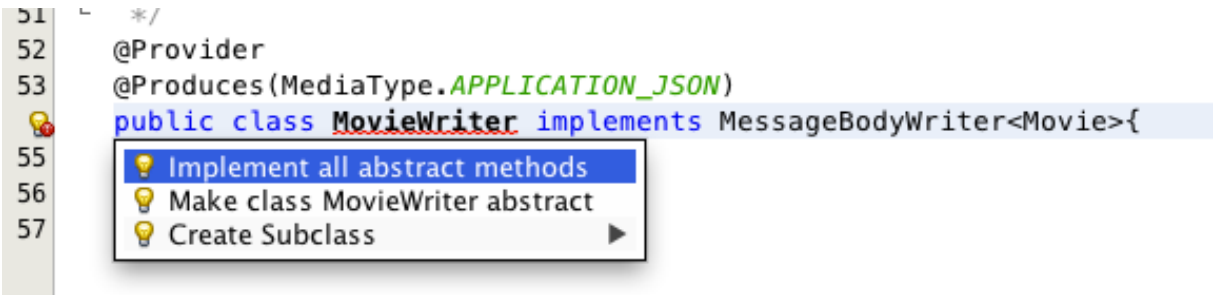


Figure 7.4.

Click on the hint (show as yellow bulb) on the class definition and select 'Implement all abstract methods'.

8. Change implementation of the `isWritable` method to:

```
return Movie.class.isAssignableFrom(type);
```

This method ascertains if the `MessageBodyWriter` supports a particular type.

9. Add implementation of the `getSize` method as:

```
return -1;
```

Originally, this method was called to ascertain the length in bytes of the serialized form of `t`. In JAX-RS 2.0, this method is deprecated and the value returned by the method is ignored by a JAX-RS runtime. All `MessageBodyWriter` implementations are advised to return -1.

10. Change implementation of the `writeTo` method to:

```
JsonGenerator gen = Json.createGenerator(entityStream);
gen.writeStartObject()
    .write("id", t.getId())
    .write("name", t.getName())
    .write("actors", t.getActors())
    .writeEnd();
    gen.flush();
```

This method writes a type to an HTTP message. `JsonGenerator` writes JSON data to an output stream in a streaming way. Overloaded write methods are used to write different data types to the stream.

Resolve the imports.

11. In 'Web Pages', right-click on 'client' folder, select 'New', 'Facelets Template Client'. Give the File Name as 'addmovie'. Click on 'Browse' next to 'Template:', expand 'Web Pages', 'WEB-INF', select 'template.xhtml', and click on 'Select File'. Click on 'Finish'.

12. In this file, remove `<ui:define>` sections where name attribute value is 'top' and 'left'. These sections are inherited from the template.
Replace `<ui:define>` section with 'content' name such that it looks like:

```
<ui:define name="content">
```

```
<h1>Add a New Movie</h1>
<h:form>
    <table cellpadding="5" cellspacing="5">
        <tr>
            <th align="left">Movie Id:</th>
            <td><h:inputText value="#{movieBackingBean.movieId}"/></td>
        </tr>
        <tr>
            <th align="left">Movie Name:</th>
            <td><h:inputText value="#{movieBackingBean.movieName}"/> </
td>
        </tr>
        <tr>
            <th align="left">Movie Actors:</th>
            <td><h:inputText value="#{movieBackingBean.actors}"/></td>
        </tr>
    </table>
    <h:commandButton
        value="Add"
        action="movies"
        actionListener="#{movieClientBean.addMovie()}"/>
</h:form>
</ui:define>
```

This code creates a form to accept input of `id`, `name`, and `actors` of a movie. These values are bound to fields in `MovieBackingBean`. The click of command button invokes the addMovie method from `MovieClientBean` and then renders 'movies.xhtml'.

Click on the hint (show as yellow bulb) to resolve the namespace prefix/URI mapping as shown.

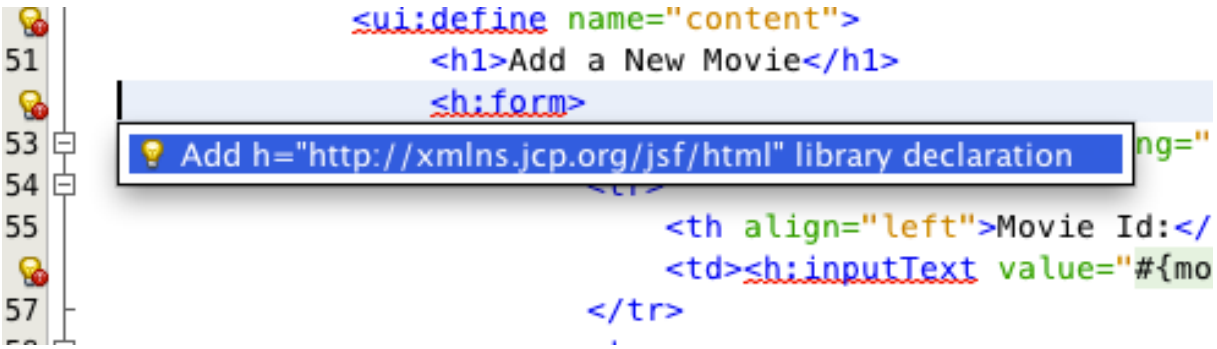


Figure 7.5.

13Add `movieName` and `actors` field to `MovieBackingBean` as:

```
String movieName;
String actors;
```

Generate getters and setters by clicking on the menu item ‘Source’ and then ‘Insert Code’.

14 Add the following code to ‘movies.xhtml’

```
<h:commandButton value="New Movie" action="addmovie" />
```

along with rest of the <commandButton>s.

15 Add the following method in `MovieClientBean`:

```
public void addMovie() {
    Movie m = new Movie();
    m.setId(bean.getMovieId());
    m.setName(bean.getMovieName());
    m.setActors(bean.getActors());
    target
        .register(MovieWriter.class)
        .request()
        .post(Entity.entity(m, MediaType.APPLICATION_JSON));
}
```

This method creates a new `Movie` instance, populates it with the values from the backing bean, and POSTs the bean to the REST endpoint. The register method registers a MovieWriter that provides conversion from the POJO to JSON. Media type of `application/json` is specified using `MediaType.APPLICATION_JSON`.

Resolve the imports as shown

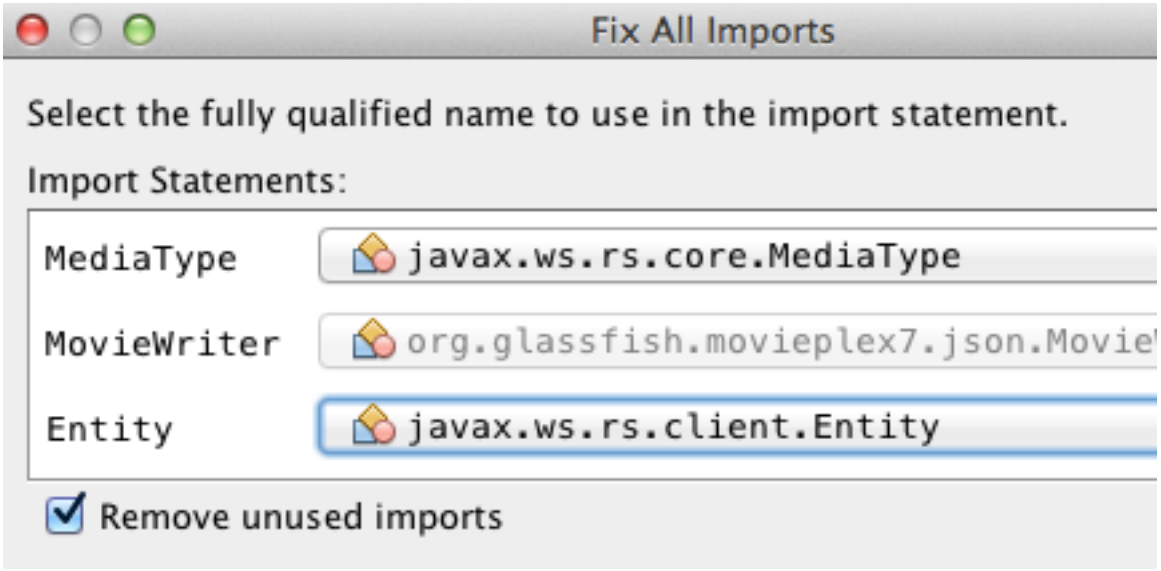


Figure 7.6.

16 Run the project to see the updated main page as:

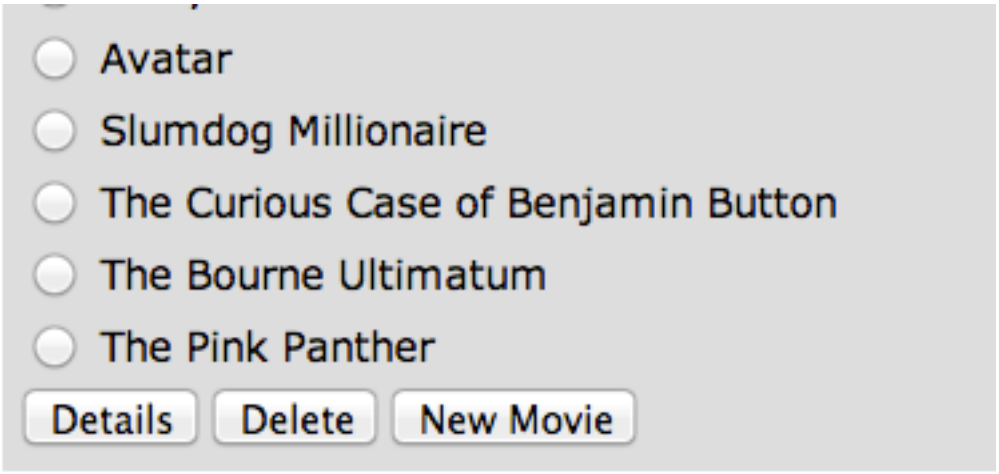


Figure 7.7.

A new movie can be added by clicking on ‘New Movie’ button.

17 Enter the details as shown:

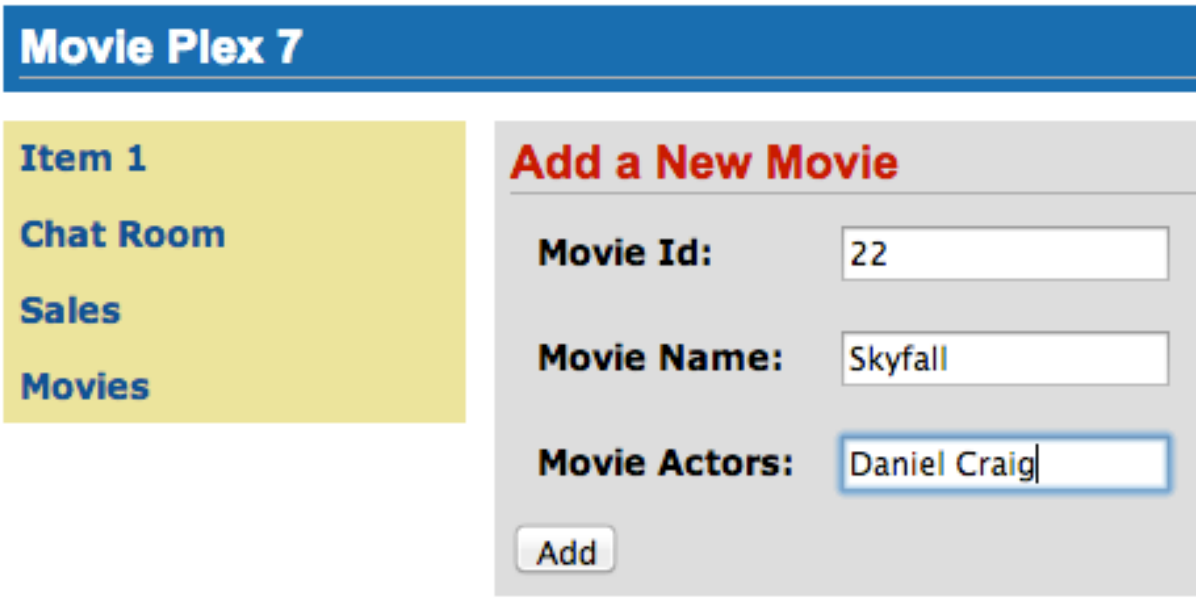


Figure 7.8.

Click on 'Add' button. The 'Movie Id' value has to be greater than 20 otherwise the primary key constraint will be violated. The table definition may be updated to generate the primary key based upon a sequence; however this is not done in the application.

The updated page looks like as shown

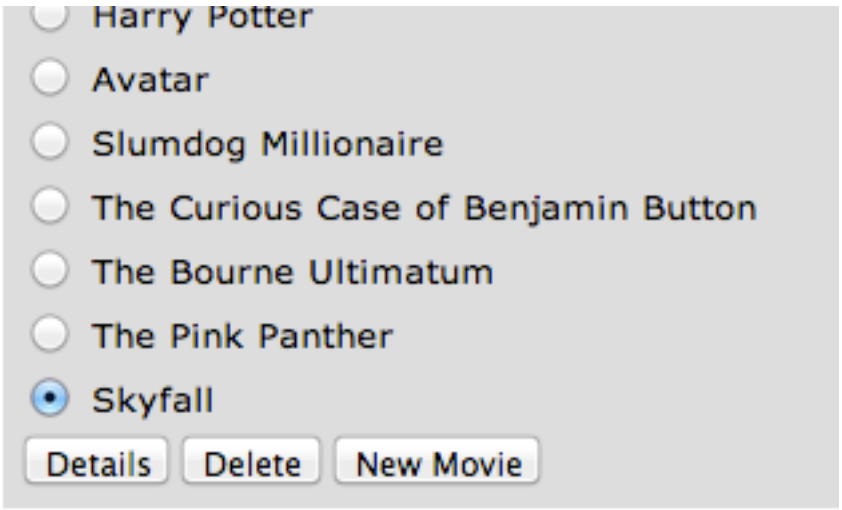


Figure 7.9.

Note that the newly added movie is now displayed.

Chapter 8. Movie Points (Java Message Service)

Purpose: Customers accrue points for watching a movie.

Estimated Time: 30-45 mins

Java Message Service 2.0 allows sending and receiving messages between distributed systems. JMS 2 introduced several improvements over the previous version such as:

- New `JMSContext` interface
- AutoCloseable `JMSContext`, `Connection`, and `Session`
- Use of runtime exceptions
- Method chaining on `JMSProducer`
- Simplified message sending

This section will provide a page to simulate submission of movie points accrued by a customer. These points are submitted to a JMS queue that is then read synchronously by another bean. JMS queue for further processing, possibly storing in the database using JPA.

1. Right-click on Source Packages, select ‘New’, ‘Java Class’, specify the name as ‘SendPointsBean’, package as ‘org.javaee7.movieplex7.points’, and click on ‘Finish’.
   Add the following class-level annotations:

   ```
   @Named
   @RequestScoped
   ```

   This makes the bean to be EL-injectable and automatically activated and passivated with the request.

   Resolve the imports.

Figure 8.1.

2. A message to a JMS Queue is sent after the customer has bought the tickets. Another bean will then retrieve this message and update the points for that customer. This allows the two systems, one generating the data about tickets purchased and the other about crediting the account with the points, completely decoupled.
   This lab will mimic the sending and consuming of a message by an explicit call to the bean from a JSF page.

   Add the following field to the class:

```
@NotNull
@Pattern(regexp = "^\\d{2},\\d{2}",
         message = "Message format must be 2 digits, comma, 2 digits,
 e.g.12,12")
private String message;
```

   This field contains the message sent to the queue. This field's value is bound to an inputText in a JSF page (created later). Constraints have been specified on this bean that enable validation of data on form submit. It requires the data to consists of two numerical digits, followed by a comma, and then two more numerical digits. If the message does not meet the validation criteria then the error message to be displayed is specified using message attribute.

   This could be thought as conveying the customer identifier and the points accrued by that customer.

Generate getter/setters for this field. Right-click in the editor pane, select 'Insert Code' (Ctrl + I shortcut on Mac), select 'Getter and Setter', select the field, and click on 'Generate'.

3. Add the following code to the class:

```
@Inject
JMSContext context;

@Resource(lookup = "java:global/jms/pointsQueue")
Queue pointsQueue;

public void sendMessage() {
    System.out.println("Sending message: " + message);
    context.createProducer().send(pointsQueue, message);
}
```

The Java EE Platform requires a pre-configured JMS connection factory under the JNDI name `java:comp/DefaultJMSConnectionFactory`. If no connection factory is specified then the pre-configured connection factory is used. In a Java EE environment, where CDI is enabled by default anyway, a container-managed `JMSContext` can be injected as:

```
@Inject
JMSContext context;
```

This code uses the default factory to inject an instance of container-managed `JMSContext`.

`JMSContext` is a new interface introduced in JMS 2. This combines in a single object the functionality of two separate objects from the JMS 1.1 API: a `Connection` and a `Session`.

When an application needs to send messages it use the `createProducer` method to create a `JMSProducer` that provides methods to configure and send messages. Messages may be sent either synchronously or asynchronously.

When an application needs to receive messages it uses one of several `createConsumer` or `createDurableConsumer` methods to create a `JMSConsumer`. A `JMSConsumer` provides methods to receive messages either synchronously or asynchronously.

All messages are then sent to a `Queue` instance (created later) identified by `java:global/jms/pointsQueue` JNDI name. The actual message is obtained from the value entered in the JSF page and bound to the message field.

Resolve the imports.

Make sure `Queue` class is imported from `javax.jms.Queue` instead of the default `java.util.Queue`.

Click on 'OK'.

4. Right-click on 'org.javaee7.movieplex7.points' package, select 'New', 'Java Class', specify the name as 'ReceivePointsBean'.
   Add the following class-level annotations:

```
@JMSDestinationDefinition(name = "java:global/jms/pointsQueue",
interfaceName = "javax.jms.Queue")
@Named
@RequestScoped
```

   This allows the bean to refered from an EL expression. It also activates and passivates the bean with the request.

   `JMSDestinationDefinition` is a new annotation introduced in JMS 2. It is used by the application to provision the required resources and allow an application to be deployed into a Java EE environment with minimal administrative configuration. This code will create Queue with the JNDI name `java:global/jms/pointsQueue`.

5. Add the following code to the class:

```
@Inject
JMSContext context;

@Resource(lookup="java:global/jms/pointsQueue")
Queue pointsQueue;

public String receiveMessage() {
    try (JMSConsumer consumer = context.createConsumer(pointsQueue)) {
        String message = consumer.receiveBody(String.class);
        System.out.println("Received message: " + message);
        return message;
    }
```

```
}
```

This code creates `JMSConsumer` in a try-with-resources block which is then used to synchronously receive a message. Note that `JMSConsumer` is created as an auto-managed resource and so is closed automatically after receiving each message. Alternatively asynchronous message delivery can also be setup using Message Driven Beans. However that is not covered in this lab.

6. Add the following method to the class:

```
public int getQueueSize() {
    int count = 0;
    try {
        QueueBrowser browser = context.createBrowser(pointsQueue);
        Enumeration elems = browser.getEnumeration();
        while (elems.hasMoreElements()) {
            elems.nextElement();
            count++;
        }
    } catch (JMSException ex) {
        ex.printStackTrace();
    }
    return count;
}
```

This code creates a `QueueBrowser` to look at the messages on a queue without removing them. It calculates and returns the total number of messages in the queue.

Make sure to resolve the import from `javax.jms.Queue`, take all other defaults.

7. Right-click on ‘Web Pages’, select ‘New’, ‘Folder’, specify the name as ‘points’, and click on ‘Finish’.
   In ‘Web Pages’, right-click on newly created folder, select ‘Facelets Template Client’, give the File Name as ‘points’. Click on ‘Browse’ next to ‘Template:’, expand ‘Web Pages’, ‘WEB-INF’, select ‘template.xhtml’, and click on ‘Select File’. Click on ‘Finish’.
8. In this file, remove `<ui:define>` sections where name attribute value is ‘top’ and ‘left’. These sections are inherited from the template.
   Replace the `<ui:define>` section with ‘content’ name such that it looks like:

```
<ui:define name="content">
    <h1>Points</h1>
    <h:form>
```

```
        Queue size:
            <h:outputText value="#{receivePointsBean.queueSize}"/><p/>
            <h:inputText value="#{sendPointsBean.message}"/>
            <h:commandButton
                value="Send Message"
                action="points"
                actionListener="#{sendPointsBean.sendMessage()}"/>
        </h:form>
        <h:form>
            <h:commandButton
                value="Receive Message"
                action="points"
                actionListener="#{receivePointsBean.receiveMessage()}"/>
        </h:form>
    </ui:define>
```

Click on the yellow bulb to resolve namespace prefix/URI mapping for `h:` prefix.

This page displays the number of messages in the current queue. It provides a text box for entering the message that can be sent to the queue. The first command button invokes `sendMessage` method from `SendPointsBean` and refreshes the page. Updated queue count, incremented by 1 in this case, is displayed. The second command button invokes `receiveMessage` method from `ReceivePointsBean` and refreshes the page. The queue count is updated again, decremented by 1 in this case.

If the message does not meet the validation criteria then the error message is displayed on the screen.

9. Add the following code in ‘template.xhtml’ along with other <outputLink>s:

```
<p/><h:outputLink
        value="${facesContext.externalContext.requestContextPath}/
faces/points/points.xhtml">
        Points
    </h:outputLink>
```

10. Run the project. The update page looks like as shown:

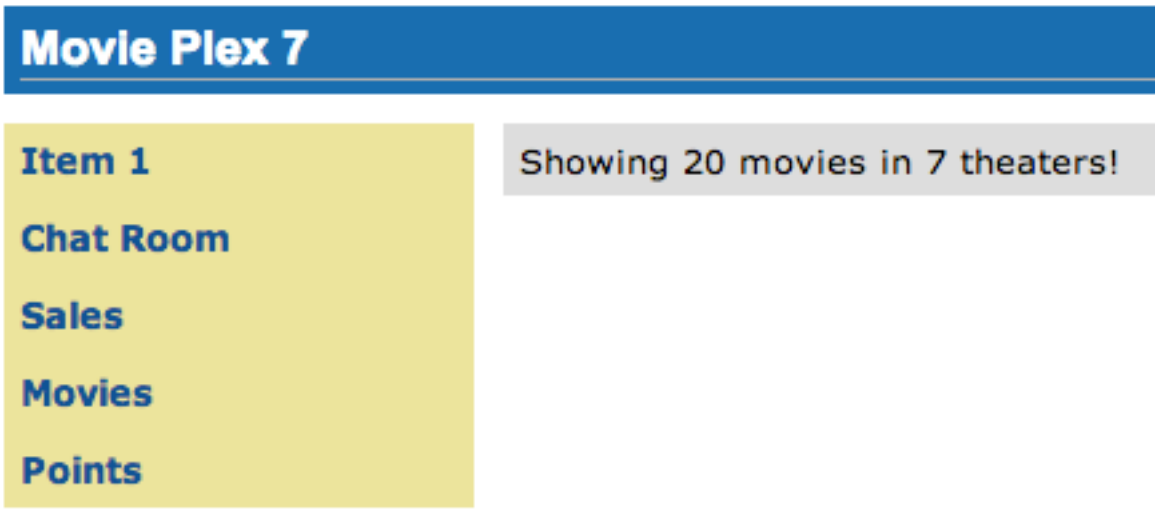


Figure 8.2.

Click on ‘Points’ to see the output as:

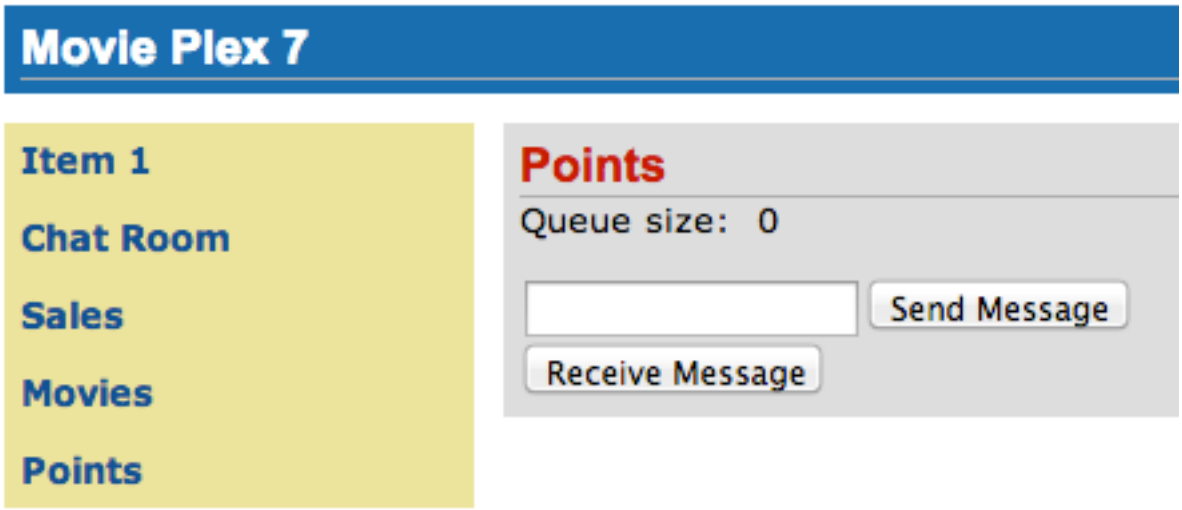


Figure 8.3.

The output shows that the queue has 0 messages. Enter a message ‘1212’ in the text box and click on ‘Send Message’ to see the output as shown.

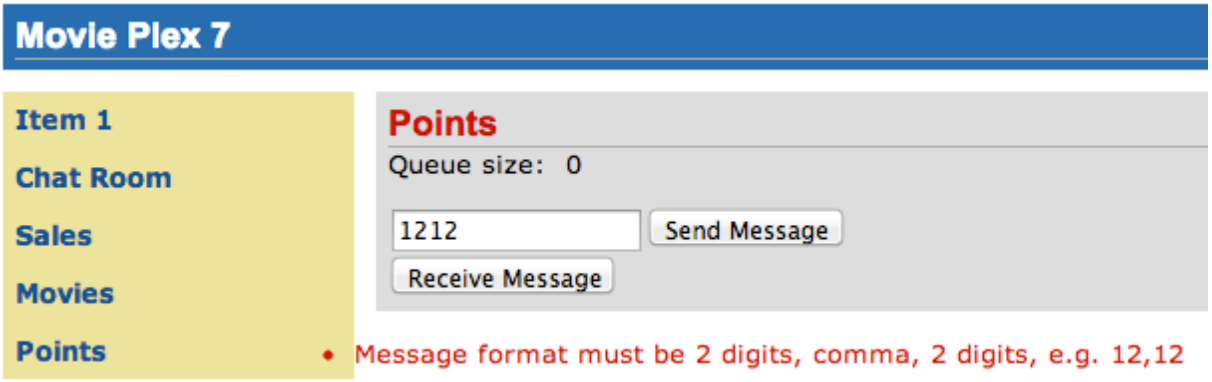


Figure 8.4.

This message is not meeting the validation criteria and so the error message is displayed.

Enter a message as '12,12' in the text box and click on 'Send Message' button to see the output as:

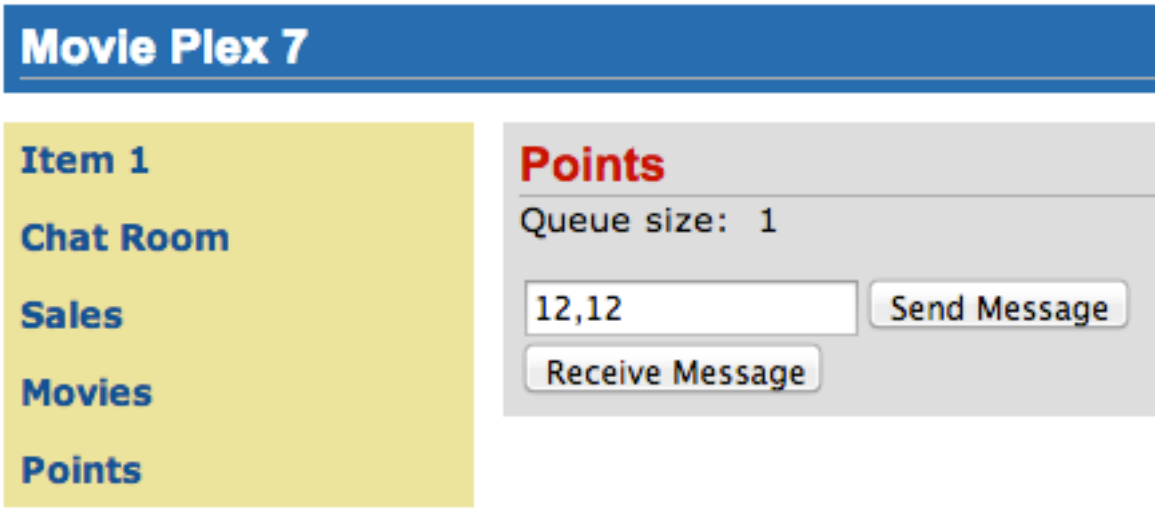


Figure 8.5.

The updated count now shows that there is 1 message in the queue. Click on 'Receive Message' button to see output as:

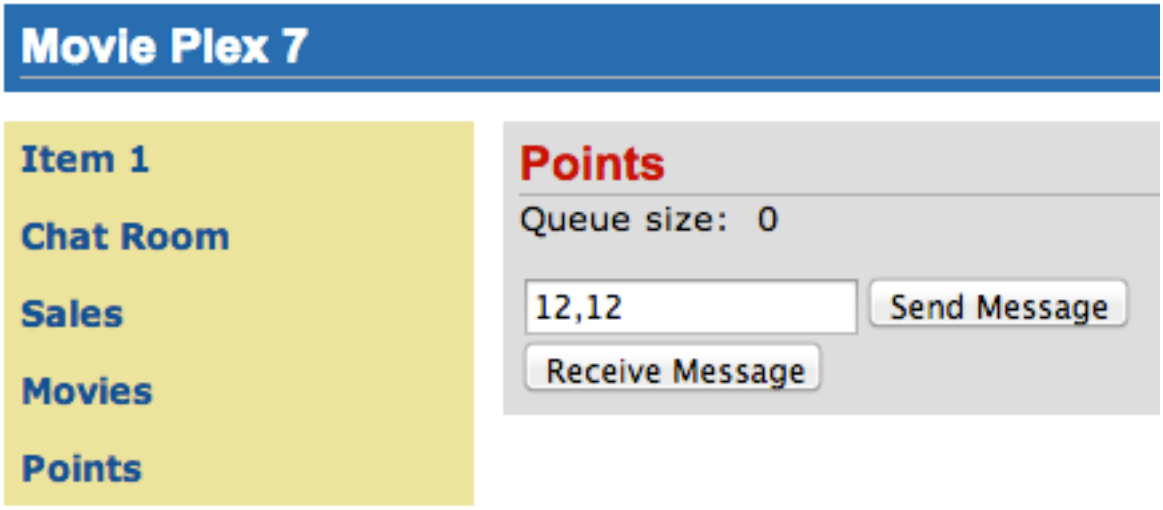


Figure 8.6.

The updated count now shows that the message has been consumed and the queue has 0 messages.

Click on ‘Send Message’ 4 times to see the output as:

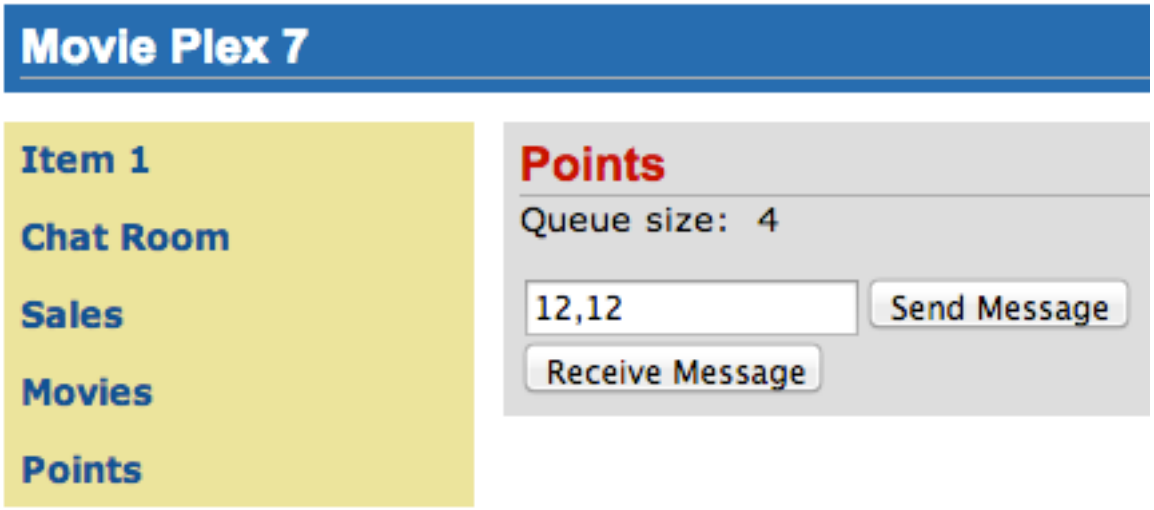


Figure 8.7.

The updated count now shows that the queue has 4 messages. Click on ‘Receive Message’ 2 times to see the output as:

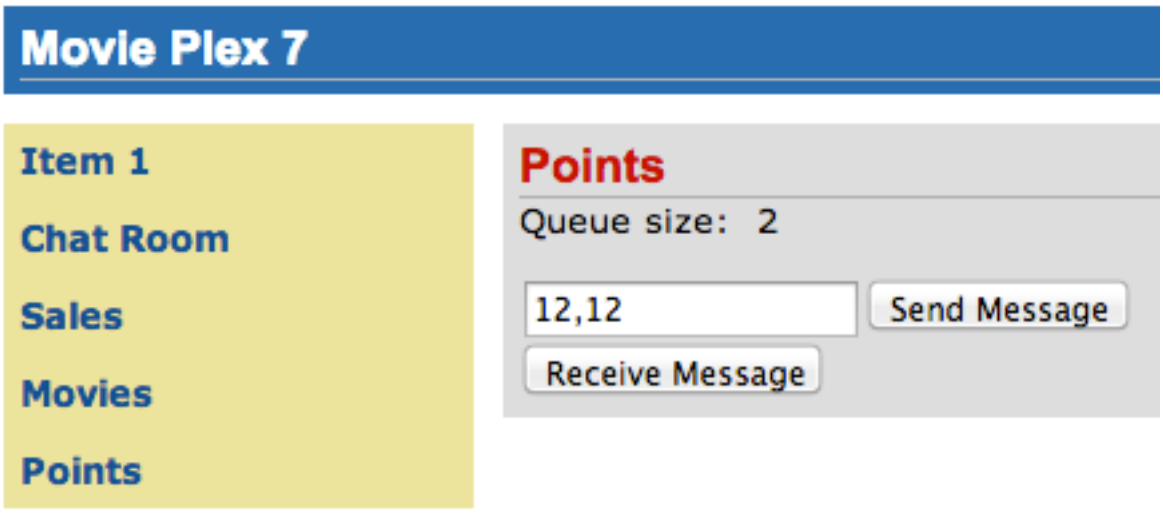


Figure 8.8.

The count is once again updated to reflect the 2 consumed and 2 remaining messages in the queue.

Chapter 9. Show Booking (JavaServer Faces)

Purpose: Build pages that allow a user to book a particular movie show in a theater. In doing so a new feature of JavaServer Faces 2.2 will be introduced and demonstrated by using in the application.

Estimated Time: 30-45 mins

JavaServer Faces 2.2 introduces a new feature called *Faces Flow* that provides an encapsulation of related views/pages with application defined entry and exit points. Faces Flow borrows core concepts from ADF TaskFlow, Spring Web Flow, and Apache MyFaces CODI.

It introduces `@FlowScoped` CDI annotation for flow-local storage and `@FlowDefinition` to define the flow using CDI producer methods. There are clearly defined entry and exit points with well-defined parameters. This allows the flow to be packaged together as a JAR or ZIP file and be reused. The application thus becomes a collection of flows and non-flow pages. Usually the objects in a flow are designed to allow the user to accomplish a task that requires input over a number of different views.

This application will build a flow that allows the user to make a movie reservation. The flow will contain four pages:

- Display the list of movies
- Display the list of available show timings
- Confirm the choices
- Make the reservation and show the ticket

Lets build the application.

1. Items in a flow are logically related to each other and so it is required to keep them together in a directory. In NetBeans, right-click on the ‘Web Pages’, select ‘New’, ‘Folder’, specify the folder name ‘booking’, and click on ‘Finish’.
2. Right-click on the newly created folder, select ‘New’, ‘Facelets Template Client’, give the File Name as ‘booking’. Click on ‘Browse’ next to ‘Template:’, expand

'Web Pages', 'WEB-INF', select 'template.xhtml', and click on 'Select File'. Click on 'Finish'.

3. 'booking.xhtml' is the entry point to the flow (more on this later).
   In this file, remove `<ui:define>` sections with 'top' and 'left' name attributes. These sections are inherited from the template.

   Replace `<ui:define>` section with 'content' name such that it looks like:

```
<ui:define name="content">
    <h2>Pick a movie</h2>
    <h:form prependId="false">
        <h:selectOneRadio
            value="#{booking.movieId}"
            layout="pageDirection"
            required="true">
            <f:selectItems
                value="#{movieFacadeREST.all}"
                var="m"
                itemValue="#{m.id}"
                itemLabel="#{m.name}"/>
        </h:selectOneRadio>
        <h:commandButton id="shows" value="Pick a
 time" action="showtimes" />
    </h:form>
</ui:define>
```

The code builds an HTML form that displays the list of movies as radio button choices. The chosen movie is bound to `#{booking.movieId}` which will be defined as a flow-scoped bean. The value of action attribute on commandButton refers to the next view in the flow, i.e. 'showtimes.xhtml' in the same directory in our case.

Click on the yellow bulb as shown and click on the suggestion to add namespace prefix/URI mapping for `h:`. Repeat the same for `f:` prefix as well.

Figure 9.1.

4. Right-click on 'Source Packages', select 'New', 'Java Class'. Specify the class name as 'Booking' and the package name as 'org.javaee7.movieplex7.booking'.
   Add `@Named` class-level annotation to make the class EL-injectable.

   Add `@FlowScoped("booking")` to define the scope of bean as the flow. The bean is automatically activated and passivated as the flow is entered or exited.

   Add the following field:

   ```
   int movieId;
   ```

   and generate getters/setters by going to 'Source', 'Insert Code', selecting 'Getter and Setter', and select the field.

   Inject `EntityManager` in this class by adding the following code:

   ```
   @PersistenceContext
   EntityManager em;
   ```

   Add the following convenience method:

   ```
   public String getMovieName() {
       try {
           return em.createNamedQuery("Movie.findById", Movie.class)
                   .setParameter("id", movieId)
   ```

```
                    .getSingleResult()
                    .getName();
        } catch (NoResultException e) {
            return "";
        }
    }
```

This method will return the movie name based upon the selected movie.

Alternatively, movie id and name may be passed from the selected radio button and parsed in the backing bean. This will reduce an extra trip to the database.

Resolve the imports.

5. Create ‘showtimes.xhtml’ in the ‘booking’ folder following the steps used to create ‘booking.xhtml’.
   In this file, remove `<ui:define>` sections with ‘top’ and ‘left’ name attributes. These sections are inherited from the template.

   Replace `<ui:define>` section with ‘content’ name such that it looks like:

```
<ui:define name="content">
    <h2>Show Timings for <font color="red">#{booking.movieName}</
font></h2>
    <h:form>

  <h:selectOneRadio value="#{booking.startTime}" layout="pageDirection" required="tr
            <c:forEach items="#{timeslotFacadeREST.all}" var="s">

  <f:selectItem itemValue="#{s.id},#{s.startTime}" itemLabel="#{s.startTime}"/
>
            </c:forEach>
        </h:selectOneRadio>
        <h:commandButton value="Confirm" action="confirm" />

  <h:commandButton id="back" value="Back" action="booking" immediate="true"/
>
    </h:form>
</ui:define>
```

This code builds an HTML form that displays the chosen movie name and all the show times. `#{timeslotFacadeREST.all}` returns the list of all the movies and iterates over them using a `c:forEach` loop. The id and start time of the selected show are bound to `#{booking.startTime}`. Command button with value ‘Back’

allows going back to the previous page and the other command button with value ‘Confirm’ takes to the next view in the flow, ‘confirm.xhtml’ in our case.

Typically a user will expect the show times only for the selected movie but all the show times are shown here. This allows us to demonstrate going back and forth within a flow if an incorrect show time for a movie is chosen. A different query may be written that displays only the shows available for this movie; however this is not part of the application.

Right-click on the yellow bulb to fix namespace prefix/URI mapping for `h:`. This needs to be repeated for `c:` and `f:` prefix as well.

6. Add the following fields to the `Booking` class:

```
String startTime;
int startTimeId;
```

And the following methods:

```
public String getStartTime() {
    return startTime;
}

public void setStartTime(String startTime) {
    StringTokenizer tokens = new StringTokenizer(startTime, ",");
    startTimeId = Integer.parseInt(tokens.nextToken());
    this.startTime = tokens.nextToken();
}

public int getStartTimeId() {
    return startTimeId;
}
```

These methods will parse the values received from the form. Also add the following method:

```
public String getTheater() {
    // for a movie and show
    try {

        // Always return the first theater
        List<ShowTiming> list =
            em.createNamedQuery("ShowTiming.findByMovieAndTimingId",
```

```
                    ShowTiming.class)
                    .setParameter("movieId", movieId)
                    .setParameter("timingId", startTimeId)
                    .getResultList();

            if (list.isEmpty())
                return "none";

            return list
                    .get(0)
                    .getTheaterId()
                    .getId()
                    .toString();
        } catch (NoResultException e) {
            return "none";
        }
    }
```

This method will find the first theater available for the chosen movie and show the timing.

Additionally a list of theaters offering that movie may be shown in a separate page.

Resolve the imports.

7. Create ‘confirm.xhtml’ page in the ‘booking’ folder by following the steps used to create ‘booking.xhtml’.
   In this file, remove `<ui:define>` sections wht ‘top’ and ‘left’ name attributes. These sections are inherited from the template.

   Replace ‘<ui:define>’ section with ‘content’ name such that it looks like:

```
<ui:define name="content">
    <c:choose>
        <c:when test="#{booking.theater == 'none'}">
            <h2>No theater found, choose a different time</h2>
            <h:form>
                Movie name: #{booking.movieName}<p/>
                Starts at: #{booking.startTime}<p/>

  <h:commandButton id="back" value="Back" action="showtimes"/>
            </h:form>
        </c:when>
        <c:otherwise>
            <h2>Confirm ?</h2>
```

```
                <h:form>
                    Movie name: #{booking.movieName}<p/>
                    Starts at: #{booking.startTime}<p/>
                    Theater: #{booking.theater}<p/>
                    <h:commandButton id="next" value="Book" action="print"/
>

  <h:commandButton id="back" value="Back" action="showtimes"/>
                </h:form>
            </c:otherwise>
        </c:choose>
</ui:define>
```

The code displays the selected movie, show timing, and theater if available. The reservation can proceed if all three are available. ‘print.xhtml’ is the last page that shows the confirmed reservation and is shown when ‘Book’ commandButton is clicked.

`actionListener` can be added to `commandButton` to invoke the business logic for making the reservation. Additional pages may be added to take the credit card details and email address.

Right-click on the yellow bulb to fix namespace prefix/URI mapping for ‘c:’. This needs to be repeated for ‘h:’ prefix as well.

8. Create ‘print.xhtml’ page in the ‘booking’ folder by following the steps used to create ‘booking.xhtml’.
   In this file, remove `<ui:define>` sections wht ‘top’ and ‘left’ name attributes. These sections are inherited from the template.

   Replace `<ui:define>` section with ‘content’ name such that it looks like:

```
<ui:define name="content">
    <h2>Reservation Confirmed</h2>
    <h:form>
        Movie name: #{booking.movieName}<p/>
        Starts at: #{booking.startTime}<p/>
        Theater: #{booking.theater}<p/>
        <h:commandButton id="home" value="home" action="goHome" /><p/>
    </h:form>
</ui:define>
```

   This code displays the movie name, show timings, and the selected theater.

Right-click on the yellow bulb to fix namespace prefix/URI mapping for 'h:'.

The `commandButton` initiates exit from the flow. The `action` attribute defines a navigation rule that will be defined in the next step.

9. 'booking.xhtml', 'showtimes.xhtml', 'confirm.xhtml', and 'print.xhtml' are all in the same directory. Now the runtime needs to be informed that the views in this directory are to be treated as view nodes in a flow. This can be done declaratively by adding 'booking/booking-flow.xml' or programmatically by having a class with a method with the following annotations:

```
@Produces @FlowDefinition
```

This lab takes the declarative approach.

Right-click on 'Web Pages/booking' folder, select 'New', 'Other', 'XML', 'XML Document', give the name as 'booking-flow', click on 'Next>', take the default of 'Well-formed Document', and click on 'Finish'.

Replace the generated code with the following:

```
<faces-config
    version="2.2"
    xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
        http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig_2_2.xsd">
    <flow-definition id="booking">
        <flow-return id="goHome">
        <from-outcome>/index</from-outcome>
        </flow-return>
    </flow-definition>
</faces-config>
```

This defines the flow graph. It uses the parent element used in a standard `faces-config.xml` but defines a `<flow-definition>` inside it.

`<flow-return>` defines a return node in a flow graph. `<from-outcome>` contains the node value, or an EL expression that defines the node, to return to. In this case, the navigation returns to the home page.

10. Finally, invoke the flow by editing 'WEB-INF/template.xhtml' and changing:

```
<h:commandLink action="item1">Item 1</h:commandLink>
```

to

```
<h:commandLink action="booking">Book a movie</h:commandLink>
```

`commandLink` renders an HTML anchor tag that behaves like a form submit button. The action attribute points to the directory where all views for the flow are stored. This directory already contains 'booking-flow.xml' which defines the flow of the pages.

11 Run the project by right clicking on the project and selecting 'Run'. The browser shows the updated output.

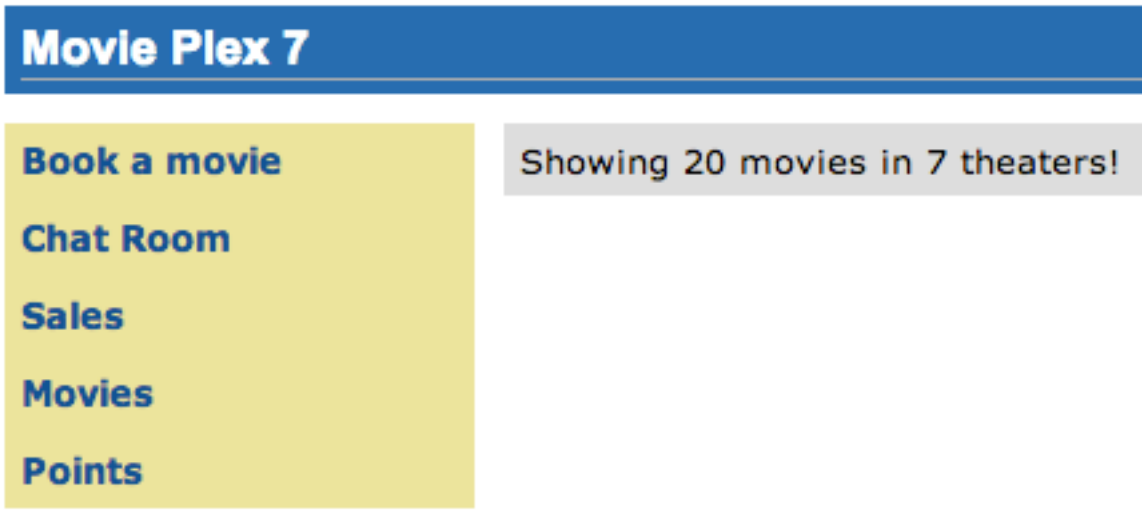


Figure 9.2.

Click on 'Book a movie' to see the page as shown.

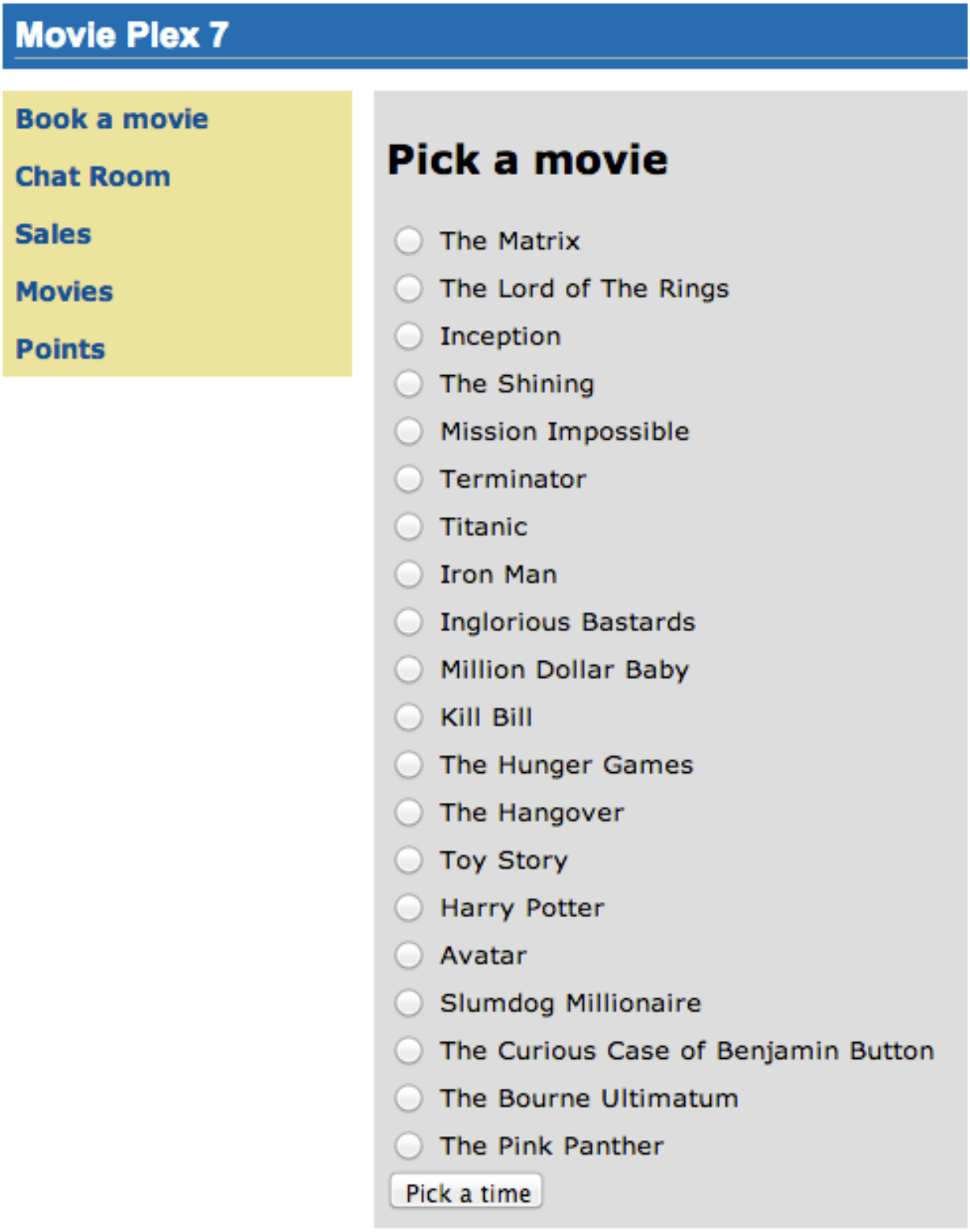


Figure 9.3.

Select a movie, say ‘The Shiningr and click on `Pick a time’ to see the page output as shown.

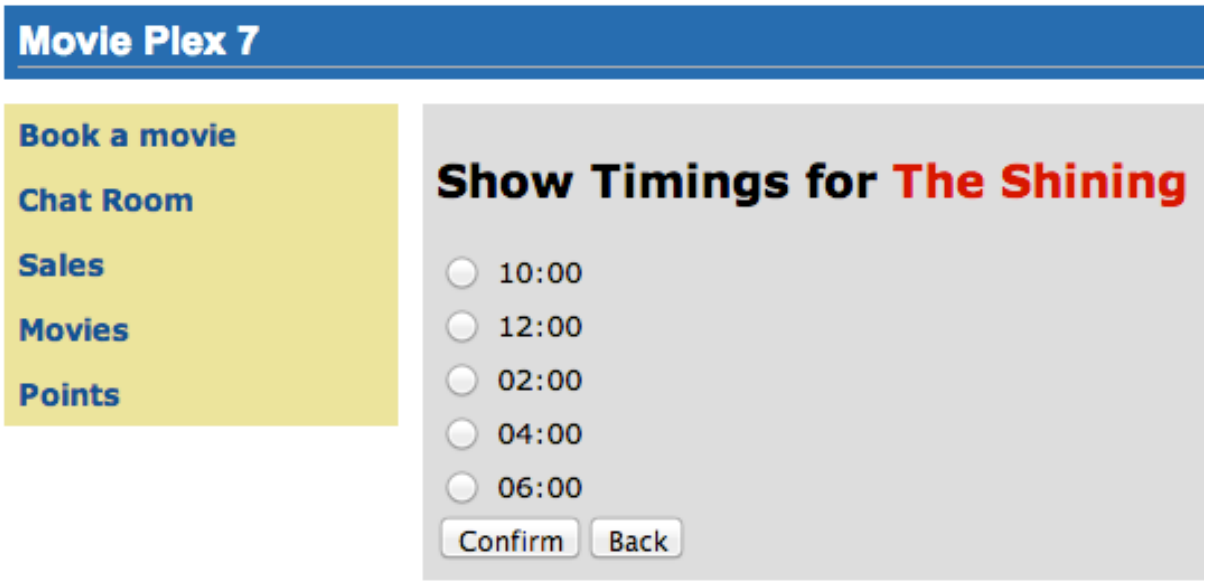


Figure 9.4.

Pick a time slot, say '04:00', click on 'Confirm' to see the output as shown.

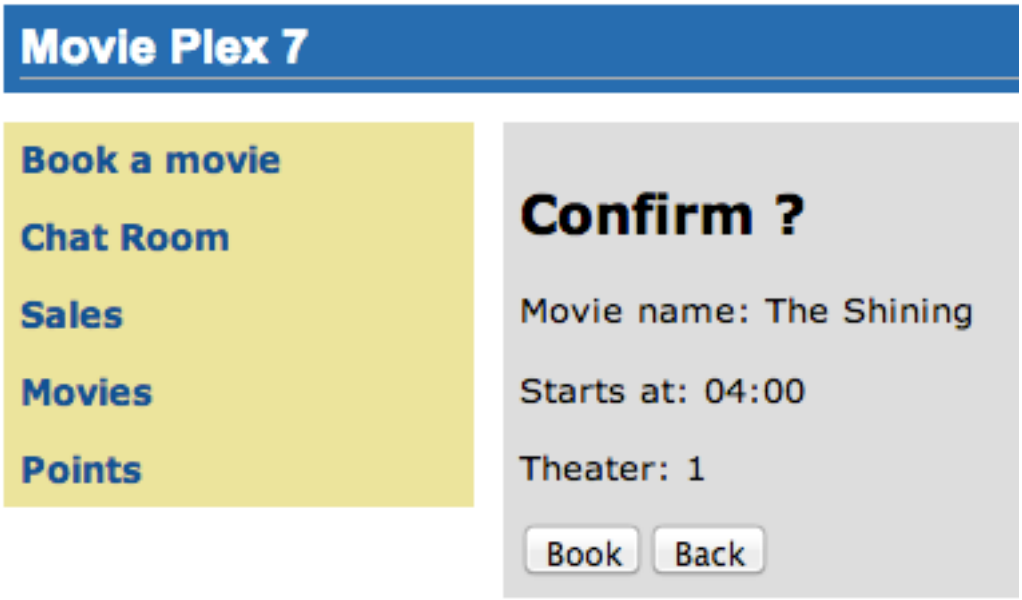


Figure 9.5.

Click on 'Book' to confirm and see the output as:

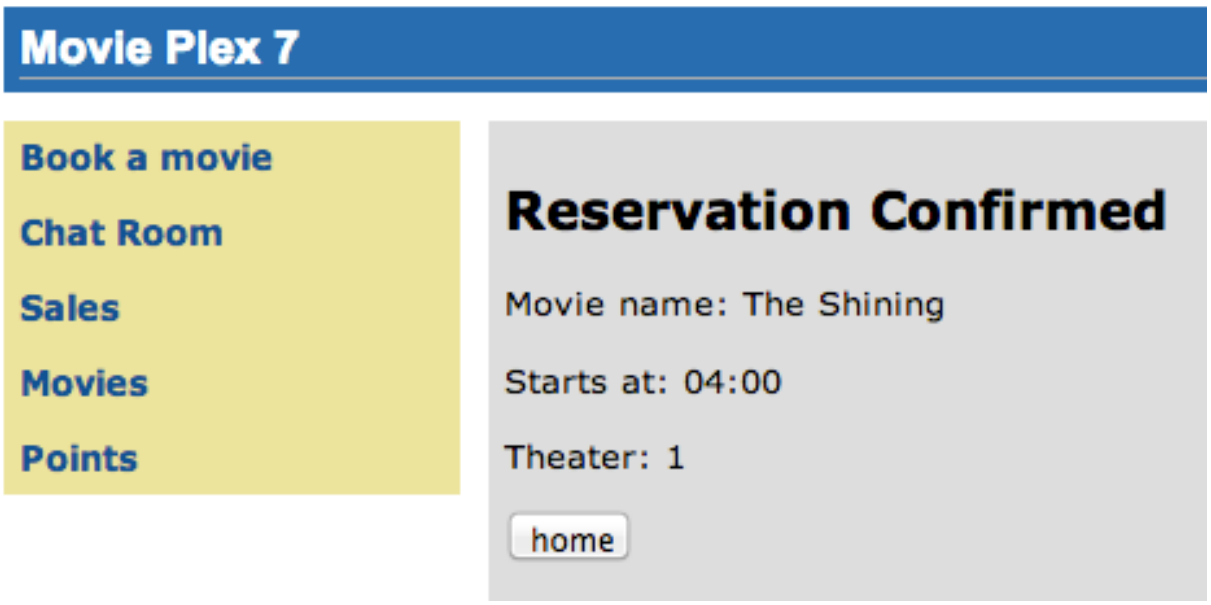


Figure 9.6.

Feel free to enter other combinations, go back and forth in the flow and notice how the values in the bean are preserved.

Click on ‘home’ takes to the main application page.

Chapter 10. Conclusion

This hands-on lab built a trivial 3-tier web application using Java EE 7 and demonstrated the following features of the platform:

1. Java EE 7 Platform
    a. Maven coordinates
    b. Default DataSource
    c. Default JMSConnectionFactory
2. Java API for WebSocket 1.0
    a. Annotated server endpoint
    b. JavaScript client
3. Batch Applications for the Java Platform 1.0
    a. Chunk-style processing
    b. Exception handling
4. Java API for JSON Processing 1.0
    a. Streaming API for generating JSON
    b. Streaming API for consuming JSON
5. Java API for RESTful Web Services 2.0
    a. Client API
    b. Custom Entity Providers
6. Java Message Service 2.0
    a. Default ConnectionFactory
    b. Injecting JMSContext
    c. Synchronous message send and receive
7. Contexts and Dependency Injection 1.1
    a. Automatic discovery of beans
    b. Injection of beans
8. JavaServer Faces 2.2
    a. Faces Flow

9. Bean Validation 1.1
    a. Integration with JavaServer Faces
10. Java Transaction API 1.2
    a. @Transactional
11. Java Persistence API 2.1
    a. Schema generation properties

Hopefully this has raised your interest enough in trying out Java EE 7 applications using WildFly 8.

Send us feedback or file issues at http://github.com/javaee-samples/javaee7-hol.

Chapter 11. Troubleshooting

11.1. How can I start/stop/restart the application server from within the IDE ?

In the ‘Services’ tab, right-click on ‘WildFly 8’. All the commands to start, stop, and restart are available from the pop-up menu.

11.2. I accidentally closed the output log window. How do I bring it back ?

In “Services” tab of NetBeans, expand ‘Servers’, choose the application server node, and select ‘View Server Log’.

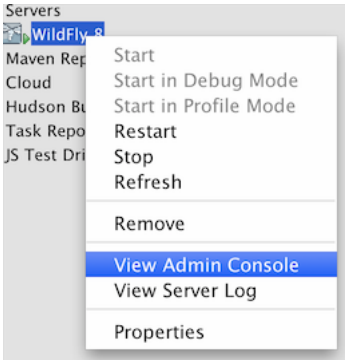


Figure 11.1.

In addition, the web-based administration console can be seen by clicking on ‘View Admin Console’.

Chapter 12. Acknowledgements

The following Java EE community members graciously reviewed and contributed to this hands-on lab:

- Antonio Goncalves (@agoncal)
- Markus Eisele (@myfear)
- Craig Sharpe (@dapugs)
- Marcus Vinicius Margarites (@mvfm)
- David Delabasse (@delabasse)
- John Clingan (@jclingan)
- Reza Rahman (@reza_rahman)
- Marian Muller (@mullermarian)
- Jason Porter (@lightguardjp)
- Dan Allen (@mojavelinux)
- Andrey Cheptsov (@andrey_cheptsov)

Thank you very much for providing the valuable feedback!

Chapter 13. Completed Solutions

The completed solution can be downloaded from javaee7-hol[1].

[1] https://github.com/javaee-samples/javaee7-hol/blob/master/solution/movieplex7-solution.zip

Chapter 14. TODO

1. Add the following use cases:
    a. Concurrency Utilities for Java EE
    b. WebSocket Java Client
2. Disable errors in persistence.xml
3. Add icons for Fix Imports, Format, Fix namespaces, Run the Project.
4. Change logging to use java.util.Logging.

Chapter 15. Revision History

1. Added IntelliJ IDEA specific instructions. (Jan 22, 2014)
2. Added macros to generate WildFly and GlassFish-server specific instructions. Also enabled IntelliJ and Eclipse specific macros. (Jan 10, 2014)
3. Moving the source document from Pages to AsciiDoc (Dec 3, 2013)

Appendix A. Appendix

A.1. Configure WildFly 8 in NetBeans

A.1.1. Configure Update Center

1. If you are using NetBeans development build then skip this section and go to Section A.1.2, “Install WildFly plugin”. Otherwise in NetBeans, click on ‘Tools’, ‘Plugins’, ‘Settings’, and click on ‘Add’.

2. Specify the name as “Dev Update Center” and the URL as “http://deadlock.netbeans.org/job/nbms-and-javadoc/lastStableBuild/artifact/nbbuild/nbms/updates.xml.gz”.

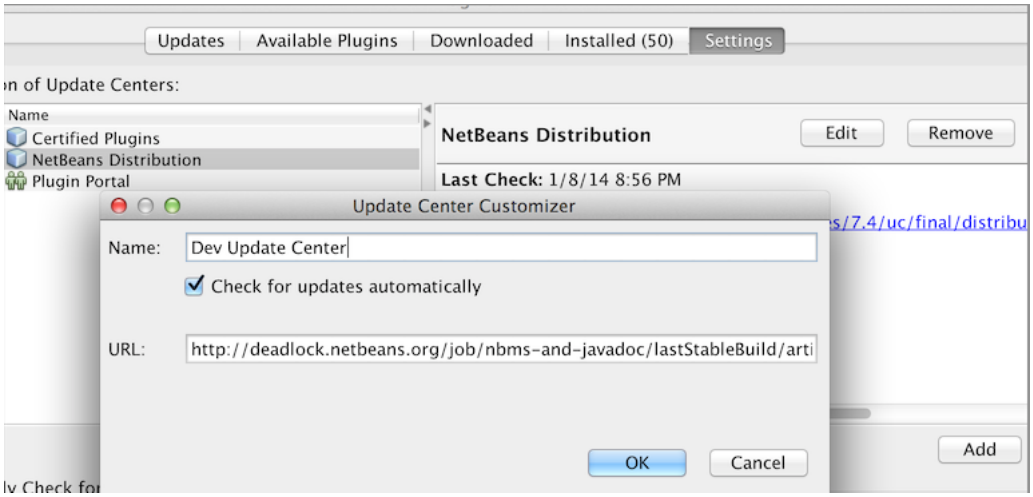


Figure A.1.

and click on ‘OK’.

A.1.2. Install WildFly plugin

1. In NetBeans, click on ‘Tools’, ‘Plugins’, ‘Available Plugins’, type “wildfly” in ‘Search:’ box, and select the plugin by clicking on the checkbox in ‘Install’ column.

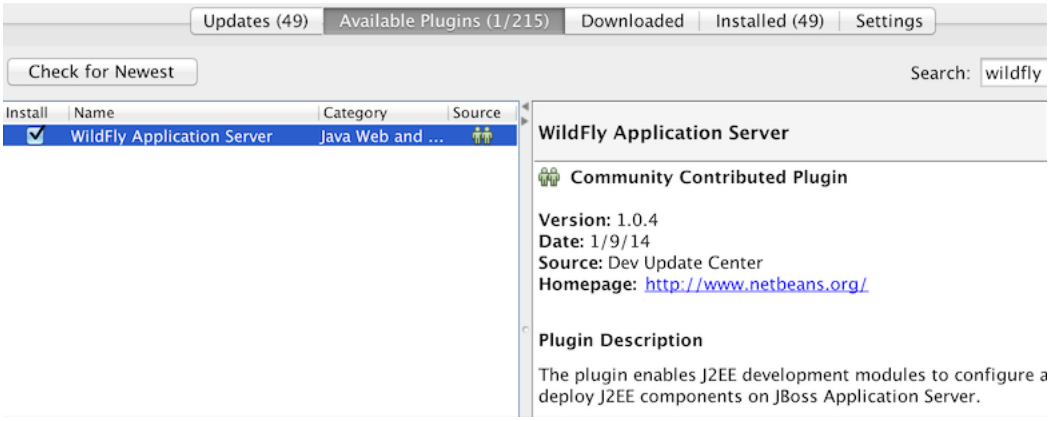


Figure A.2.

The exact plugin version and the date may be different.

2. Click on ‘Install’ button, ‘Next >’, accept the license agreement by clicking on the checkbox, and click on ‘Install’ button to install the plugin. Click on ‘Finish’ to restart the IDE and complete installation.

A.1.3. Configure WildFly 8

1. In NetBeans, click on ‘Services’ tab.
2. Right-click on Servers, choose ‘Add Server…’ in the pop-up menu.

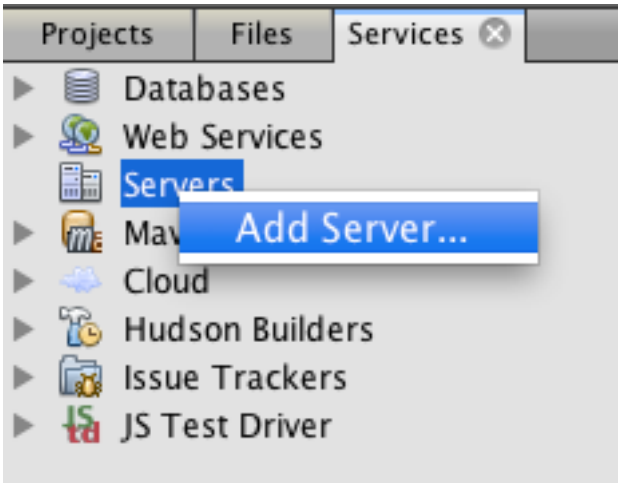


Figure A.3.

3. Select ‘WildFly Application Server’ in the Add Server Instance wizard, set the name to ‘WildFly 8’ and click ‘Next >’.

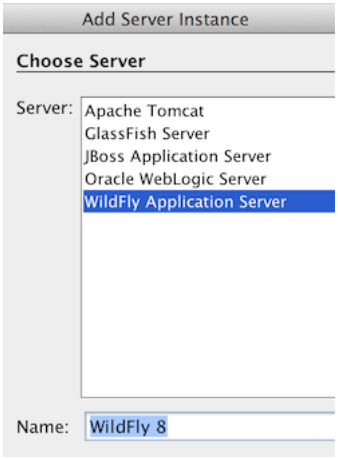


Figure A.4.

4. Click on ‘Browse’ for ‘Server Location’ and select the directory that got created when WildFly archive was unzipped. Click on ‘Browse’ for ‘Server Configuration’ and select the ‘standalone/configuration/standalone-full.xml’ file in the unzipped WildFly archive.

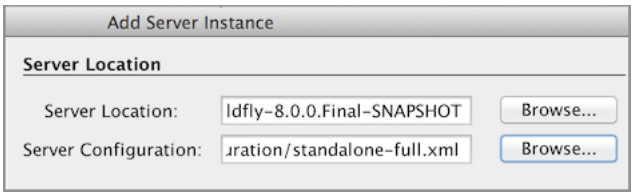


Figure A.5.

Click on ‘Next’ and then ‘Finish’. The ‘Services’ should show the WildFly instance.

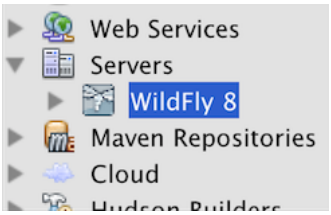


Figure A.6.

A.2. Prepare IntelliJ IDEA for working with WildFly 8

To be able to perform the exercises discussed in this tutorial, you need the Ultimate Edition of IntelliJ IDEA. Keep that in mind when downloading IntelliJ IDEA from http://www.jetbrains.com/idea/download/.

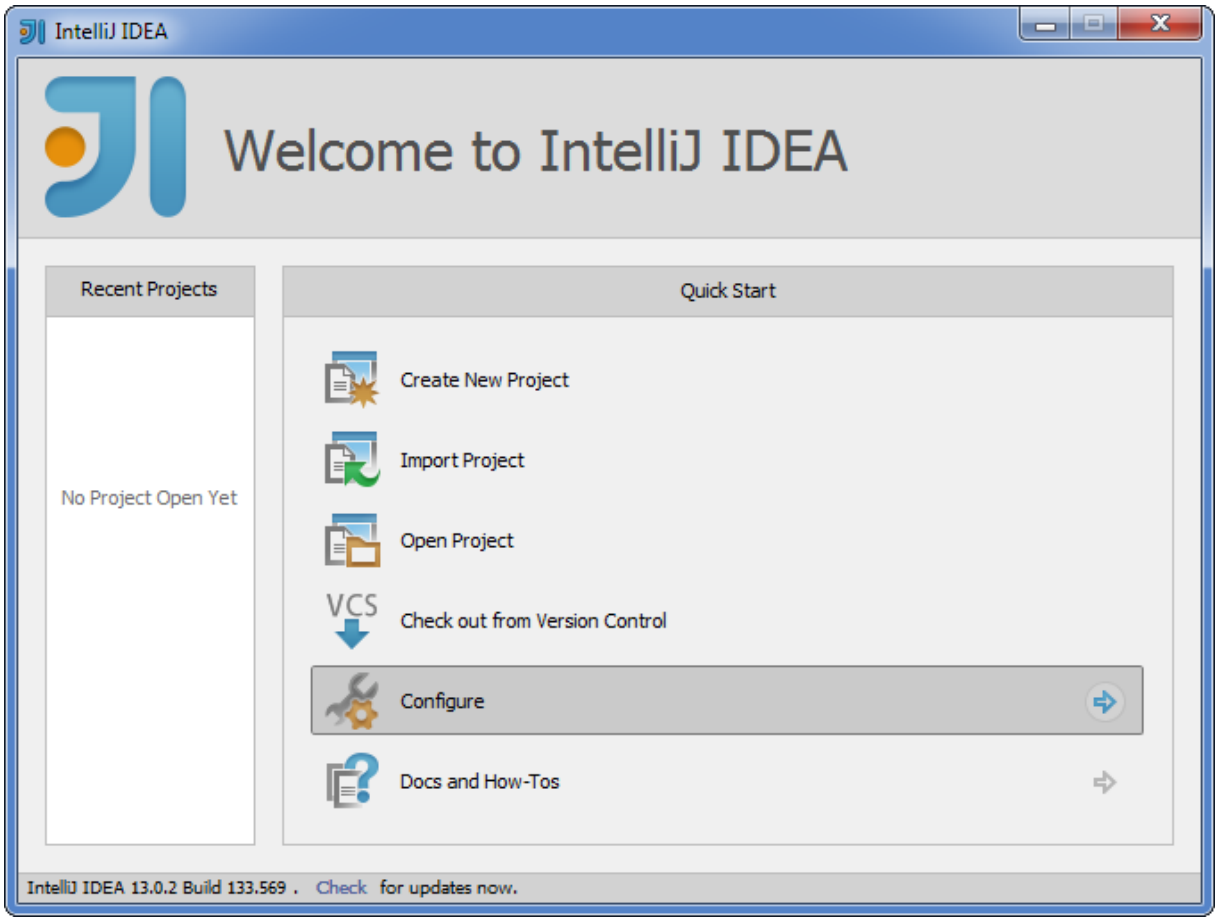
When the appropriate edition of IntelliJ IDEA is installed, you can start preparing the IDE for the exercises:

1. Section A.2.1, “Specify the JDK”
2. Section A.2.2, “Define WildFly”
3. Section A.2.3, “Create a project”
4. Section A.2.4, “Create a run/debug configuration”
5. Section A.2.5, “Run the application”

A.2.1. Specify the JDK

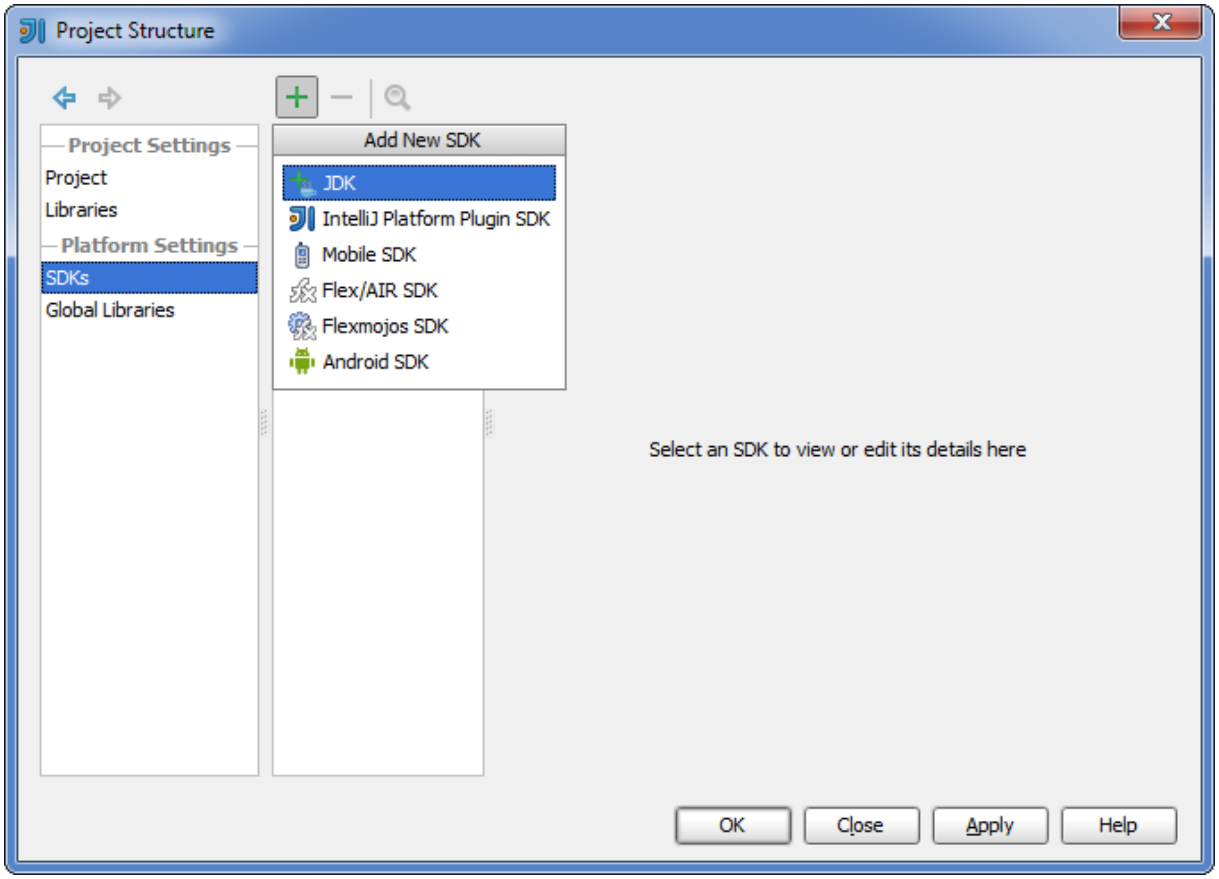
First of all, you should specify the JDK that you are going to use. In IntelliJ IDEA, this is done in the **Project Structure** dialog:

1. Start IntelliJ IDEA. If, as a result, a project opens, close the project (**File | Close Project**).
2. On the Welcome screen, under **Quick Start**, click **Configure**.

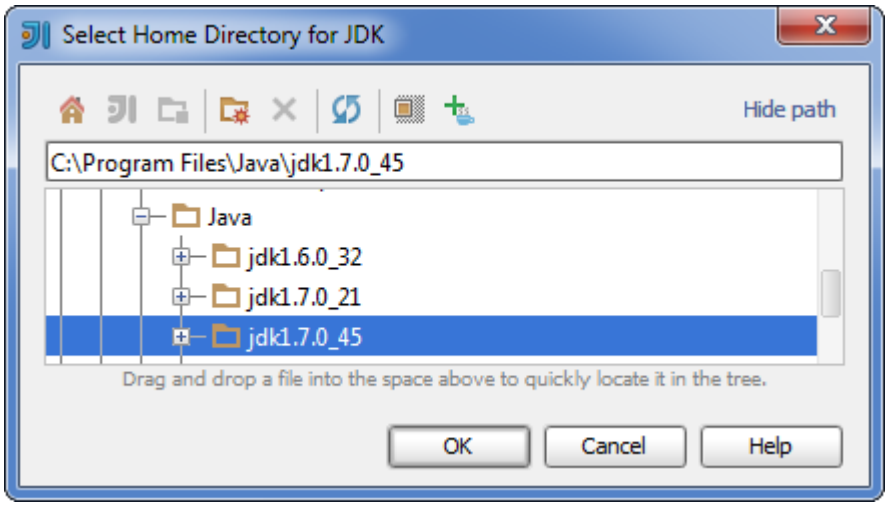


3. Under **Configure**, click **Project Defaults**, and then, under **Project Defaults**, click **Project Structure**.

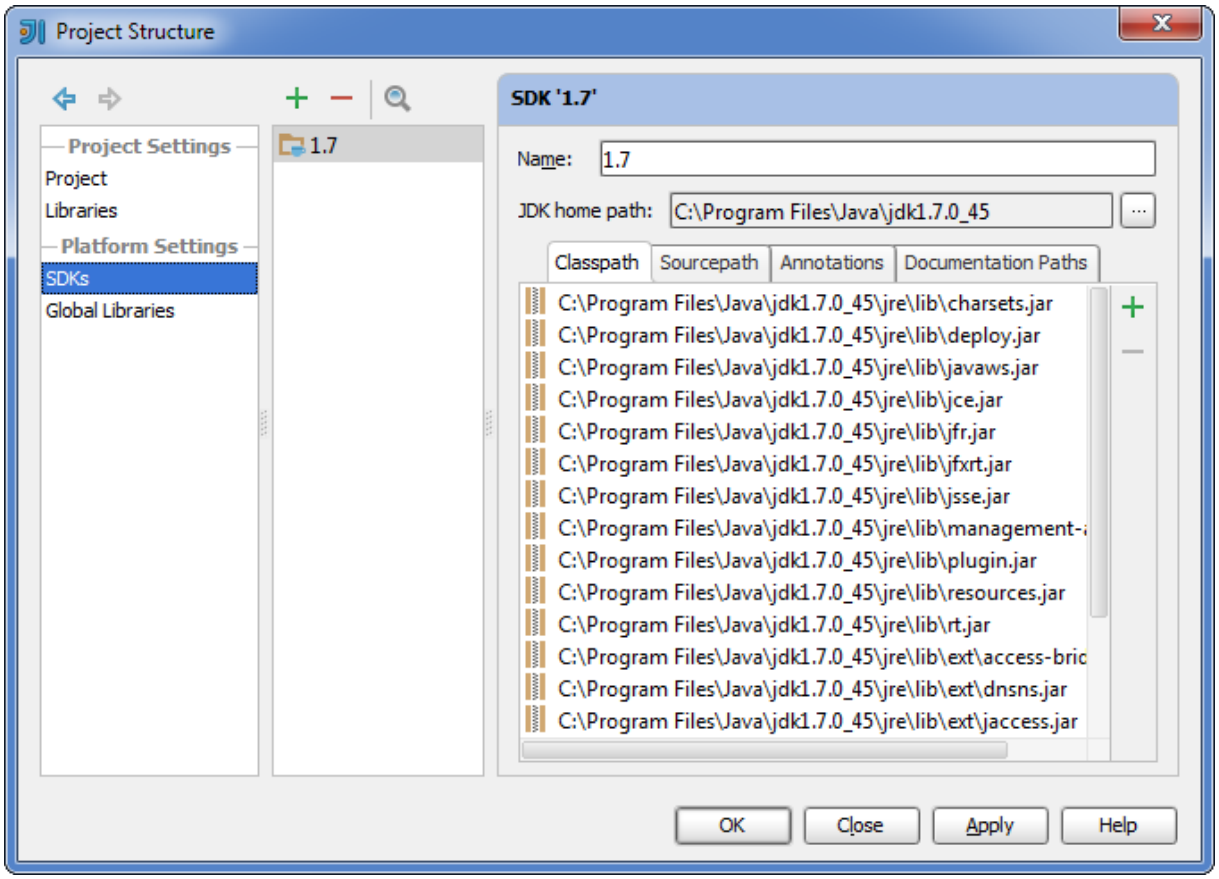
4. In the left-hand pane of the **Project Structure** dialog, under **Platform Settings**, select **SDKs**. Click +
and select **JDK**.



5. In the **Select Home Directory for JDK** dialog, select the folder in which the JDK that you are going to use is installed, and click **OK**.

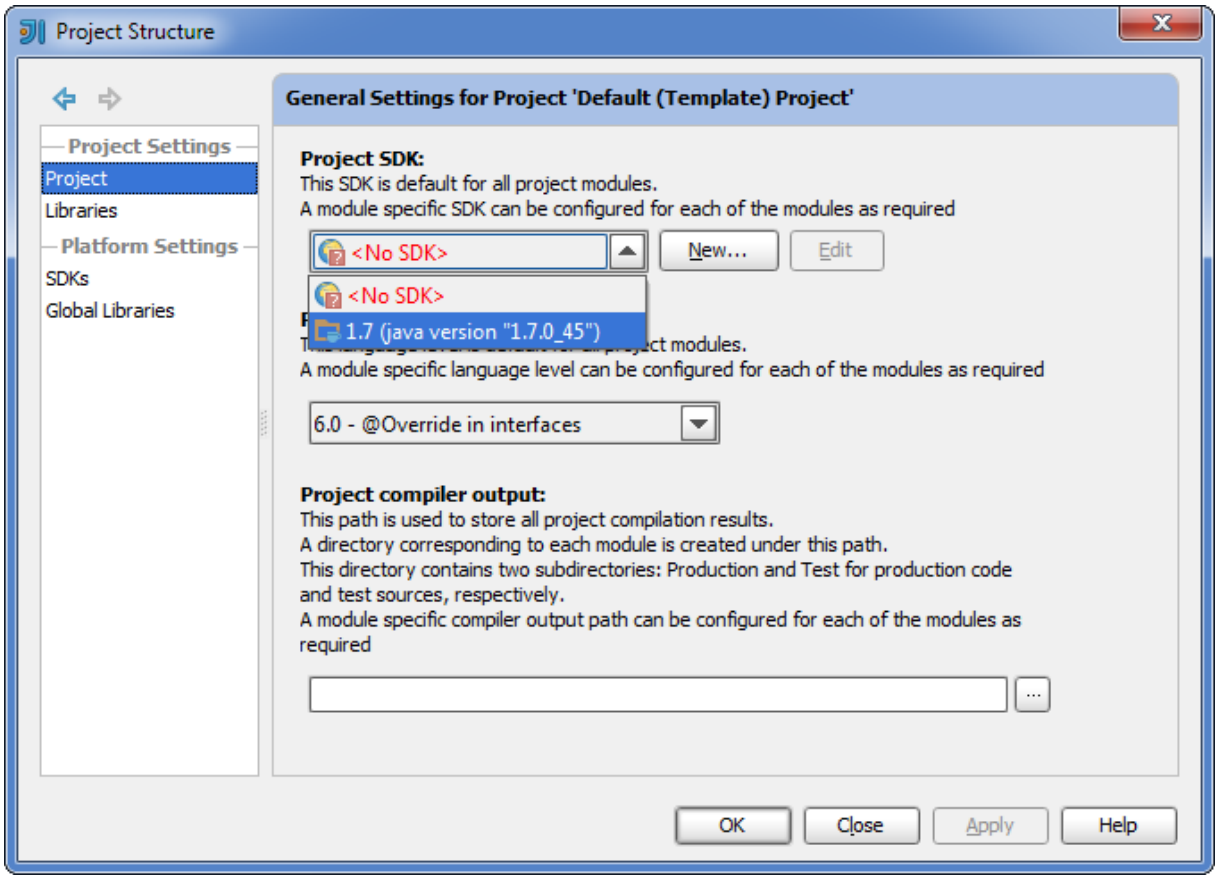


6. In the **Project Structure** dialog, click **Apply**.



Now, let's make the JDK that we have specified the default SDK.

7. In the left-hand pane, under **Project Settings**, select **Project**. In the right-hand part of the dialog, under **Project SDK**, select the JDK from the list.

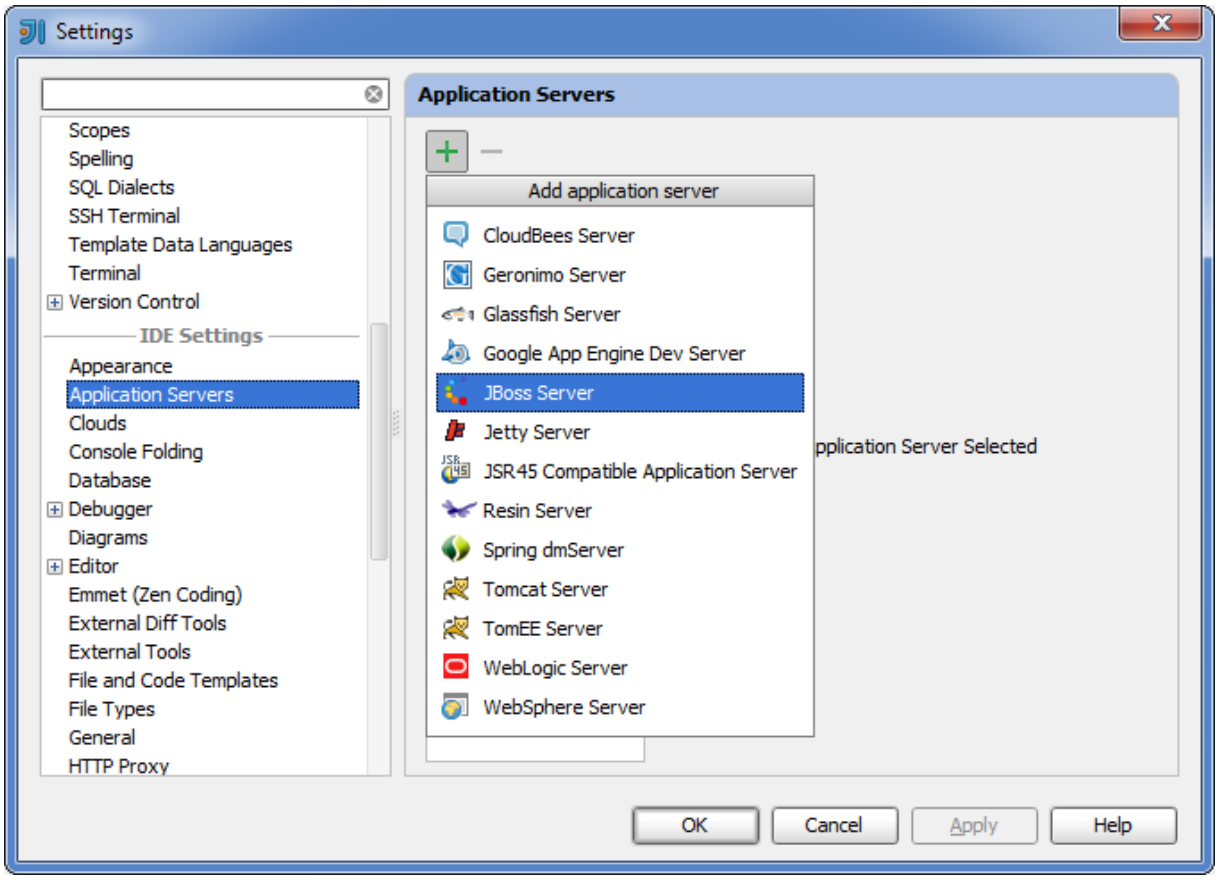


8. Click **OK**.

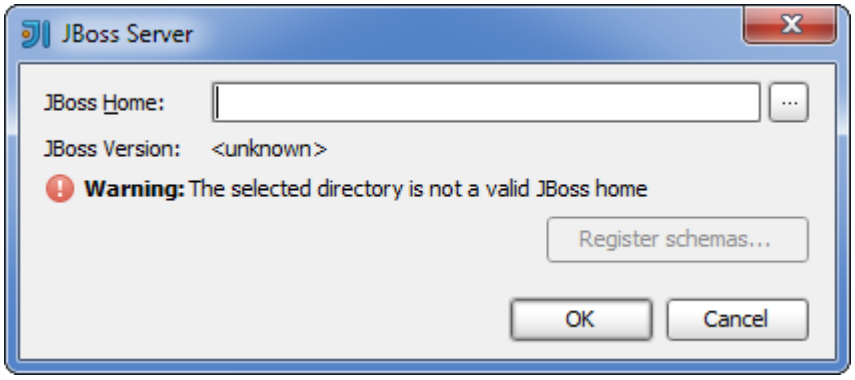
A.2.2. Define WildFly

Defining an application server in IntelliJ IDEA, normally, is just telling the IDE where the server is installed. The servers are defined in the **Settings** dialog. (On Mac OS, this dialog is called **Preferences**.)

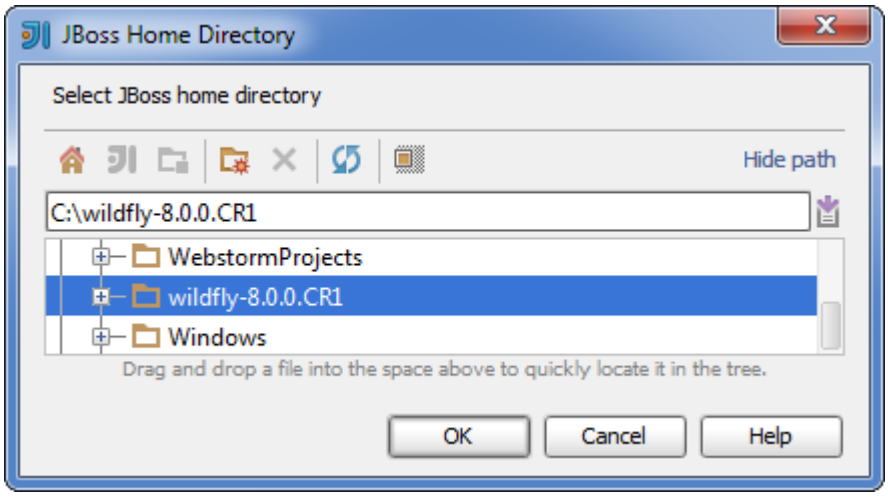
1. On the Welcome screen, to the left of **Project Defaults**, click **Back** .
2. Under **Configure**, click **Settings**.
3. In the left-hand pane of the **Settings** (**Preferences**) dialog, under **IDE Settings**, select **Application Servers**. On the **Application Servers** page, click +
   and select **JBoss Server**. (WildFly is a server from the "JBoss family".)



4. In the **JBoss Server** dialog, click ⋯ to the right of the **JBoss Home** field.

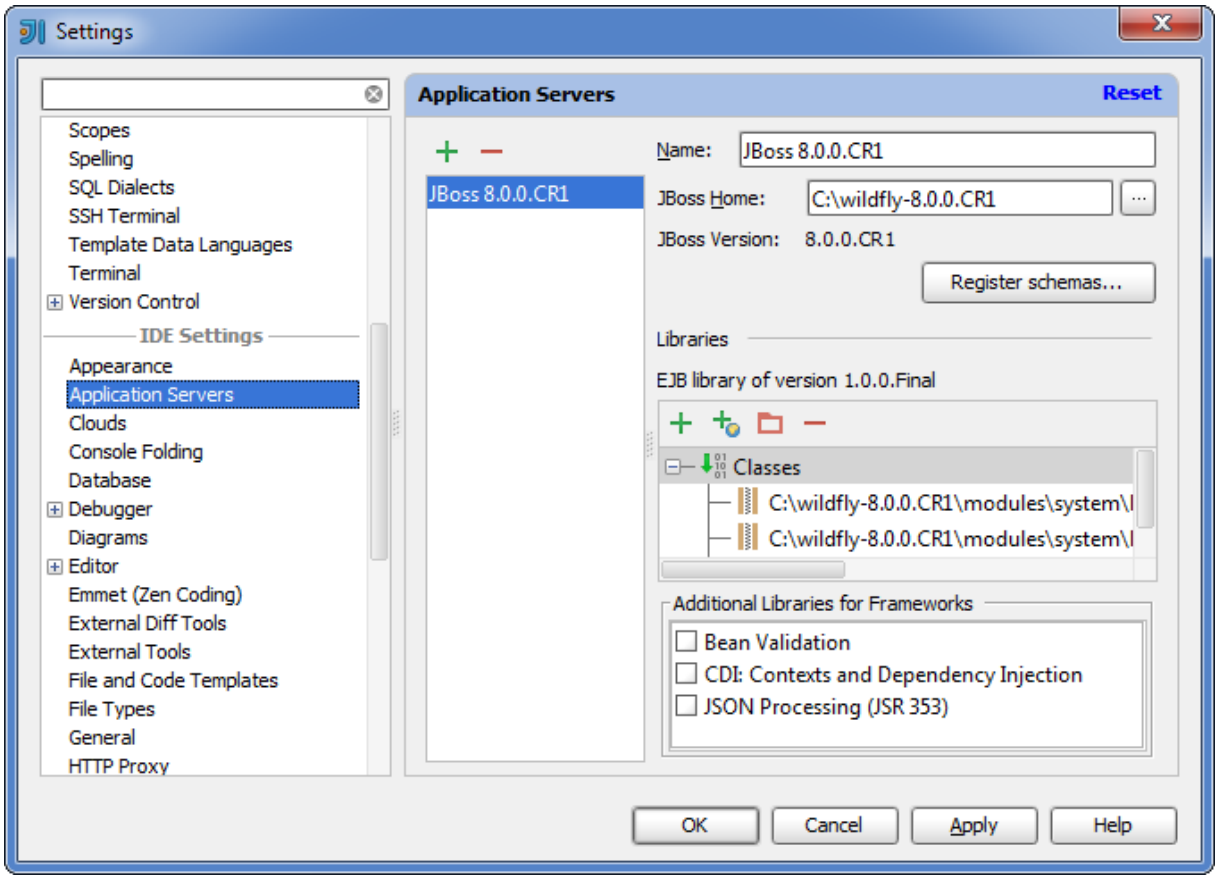


5. In the **JBoss Home Directory** dialog, select the folder in which you have the WildFly server installed, and click **OK**.



6. Click **OK** in the **JBoss Server** dialog.

7. In the **Settings** (**Preferences**) dialog, click **OK**.

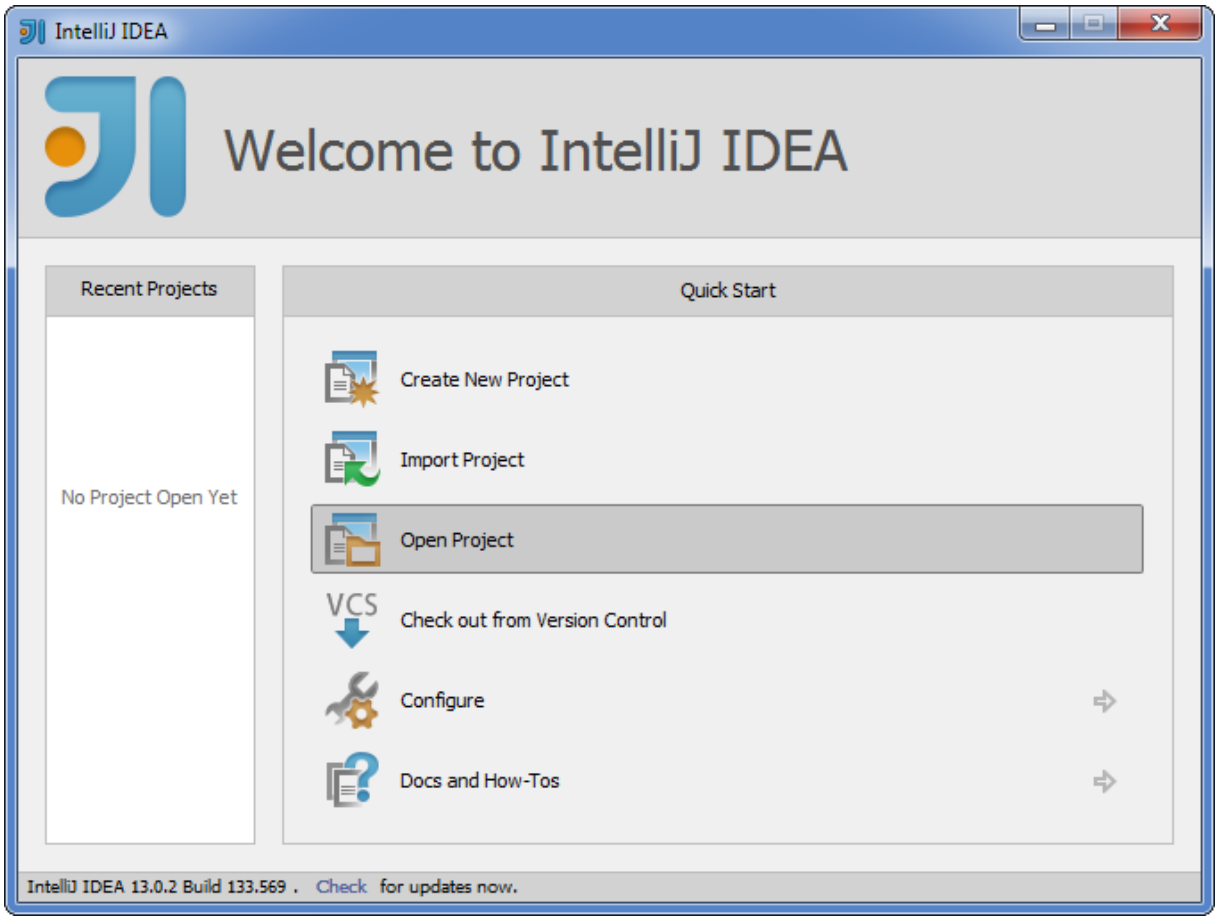


A.2.3. Create a project

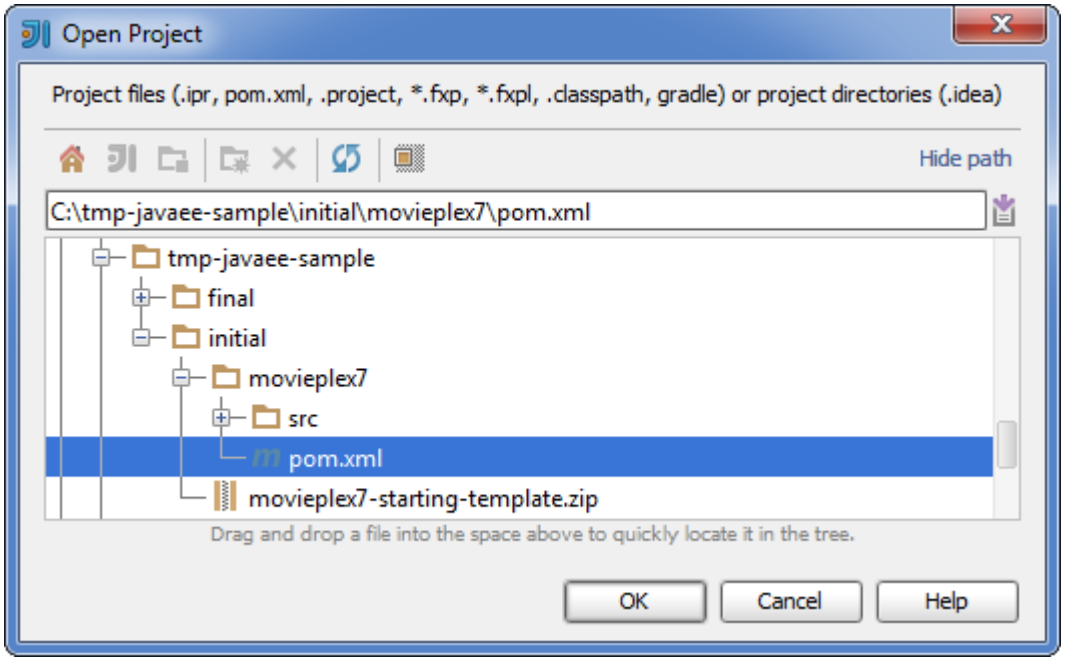
The sample application is supplied as a Maven project with an associated pom.xml[1] file that contains all the necessary project definitions. The corresponding IntelliJ IDEA project in such a case can be created by simply "opening" the `pom.xml` file. (Obviously, this isn't the only way to create projects in IDEA. You can create projects for existing collections of source files, import Eclipse and Flash Builder projects, and Gradle build scripts. Finally, you can create projects from scratch.)

1. On the Welcome screen, to the left of **Configure**, click **Back** .
2. Under **Quick Start**, click **Open Project**.

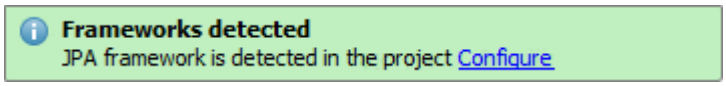
[1] http://maven.apache.org/pom.html



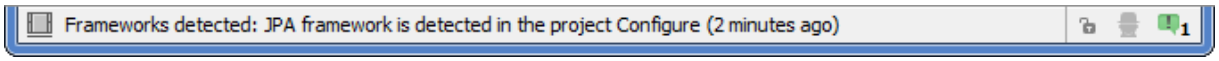
3. In the **Open Project** dialog, select the `pom.xml` file associated with the sample application, and click **OK**.



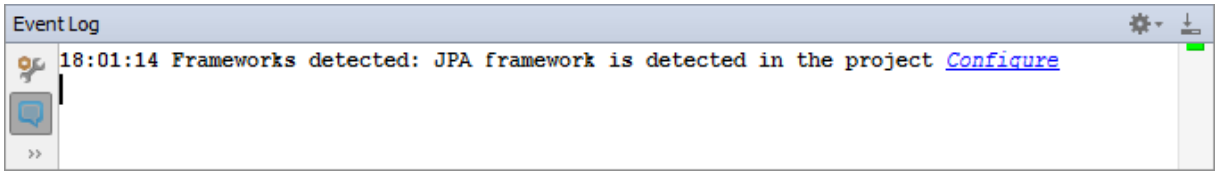
Wait while IntelliJ IDEA is processing `pom.xml` and creating the project. When this process is complete, the following message is shown:



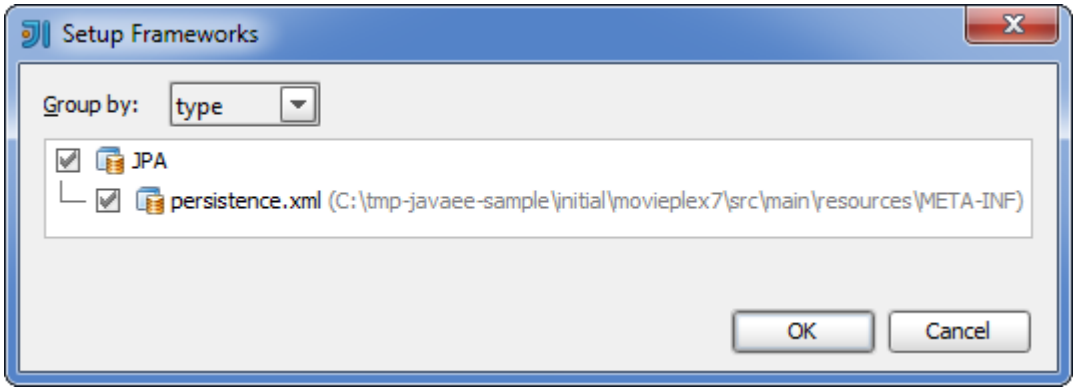
4. Click **Configure** in the message box. (If by now the message has disappeared, click  on the Status bar.



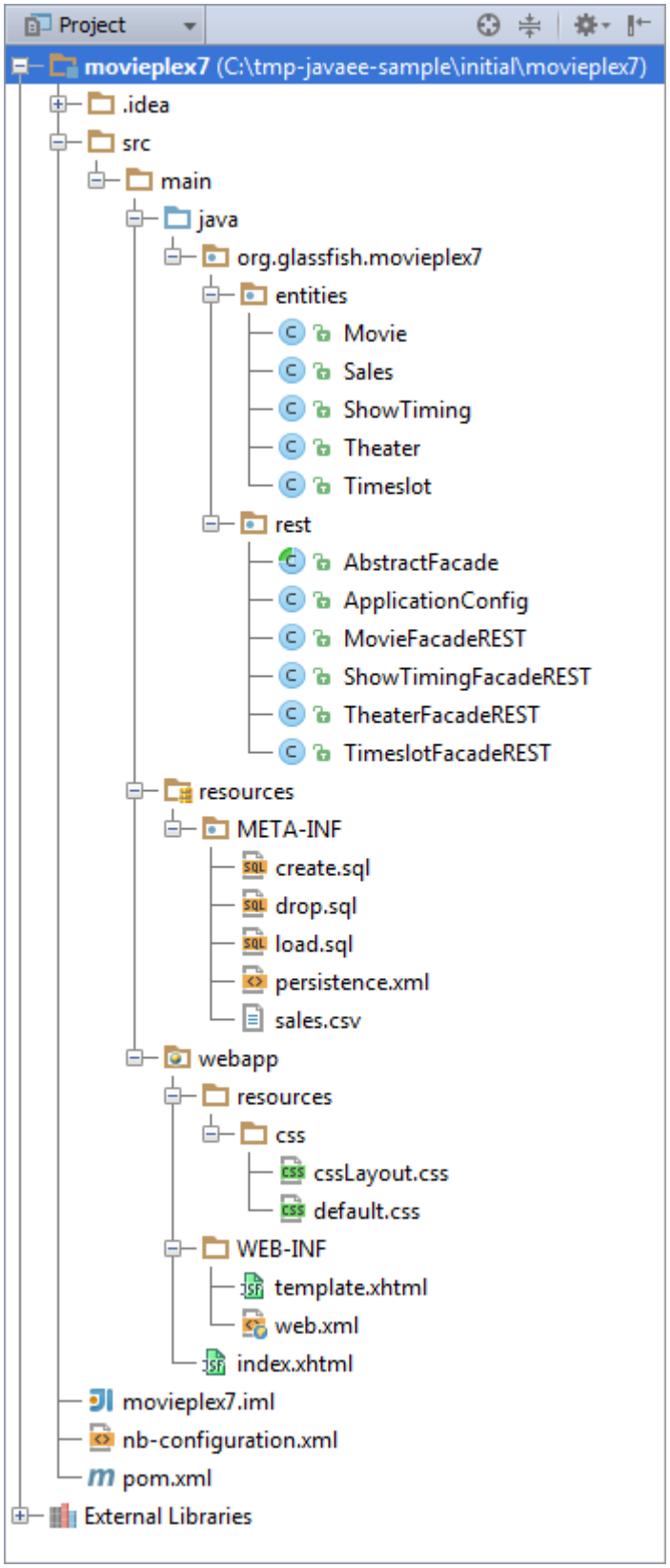
The **Event Log** tool window will open. Click **Configure** in this window.)



5. In the **Setup Frameworks** dialog, just click **OK**. (By doing so you confirm that the file `persistence.xml` found in the project belongs to the JPA framework.)



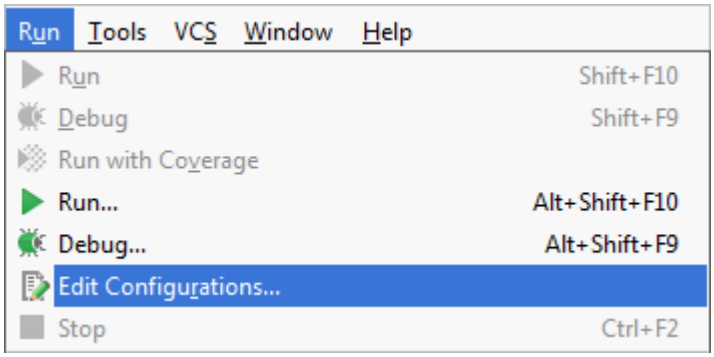
Now, as an intermediate check, make sure that the project structure looks something similar to this:


Project
movieplex7 (C:\tmp-javaee-sample\initial\movieplex7)
.idea
src
main
java
org.glassfish.movieplex7
entities
Movie
Sales
ShowTiming
Theater
Timeslot
rest
AbstractFacade
ApplicationConfig
MovieFacadeREST
ShowTimingFacadeREST
TheaterFacadeREST
TimeslotFacadeREST
resources
META-INF
create.sql
drop.sql
load.sql
persistence.xml
sales.csv
webapp
resources
css
cssLayout.css
default.css
WEB-INF
template.xhtml
web.xml
index.xhtml
movieplex7.iml
nb-configuration.xml
pom.xml
External Libraries

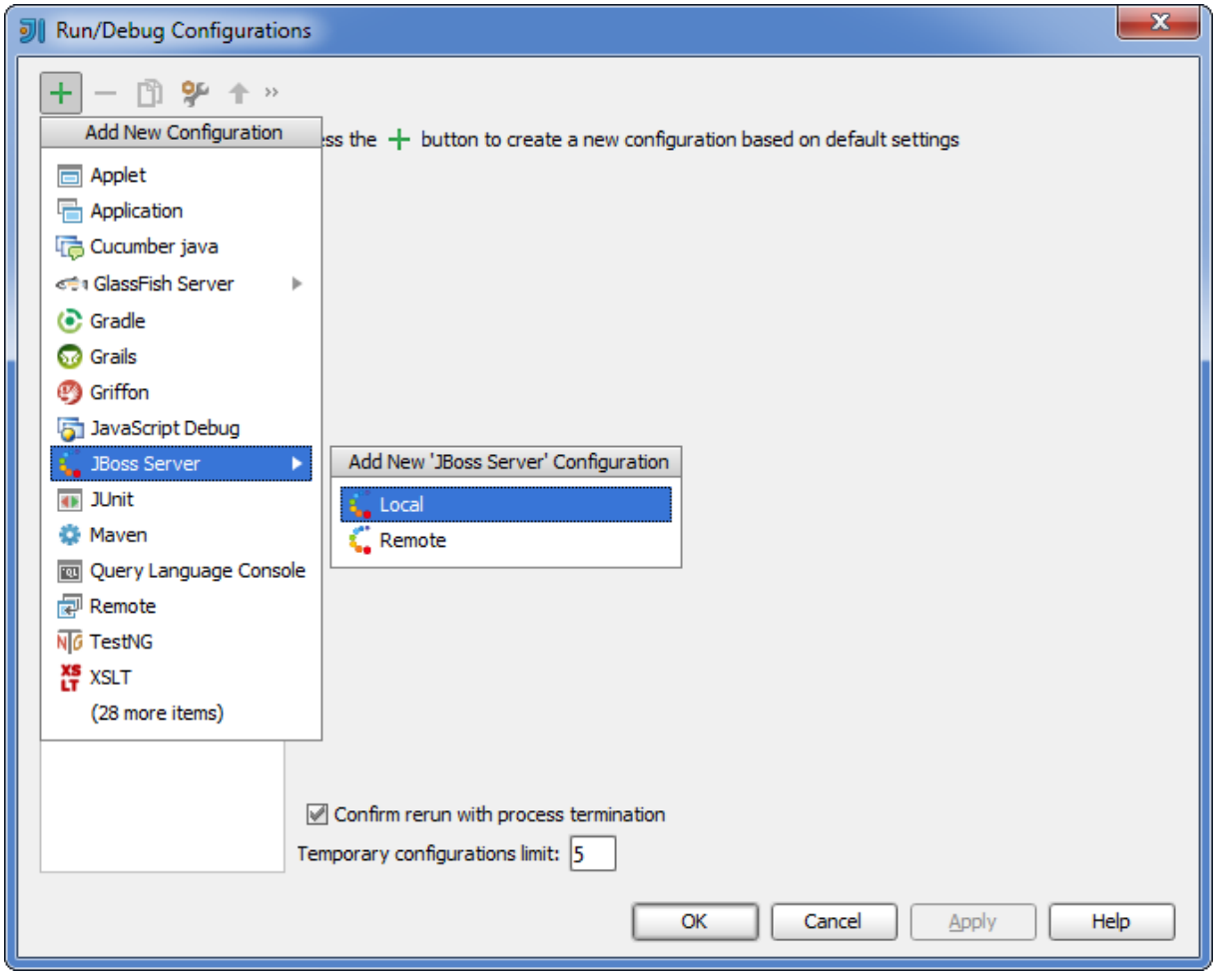
A.2.4. Create a run/debug configuration

Applications in IntelliJ IDEA are run and debugged according to what is called run/debug configurations. Now we are going to create the configuration for running and debugging the sample application in the context of WildFly.

1. In the main menu, select **Run | Edit Configurations**.



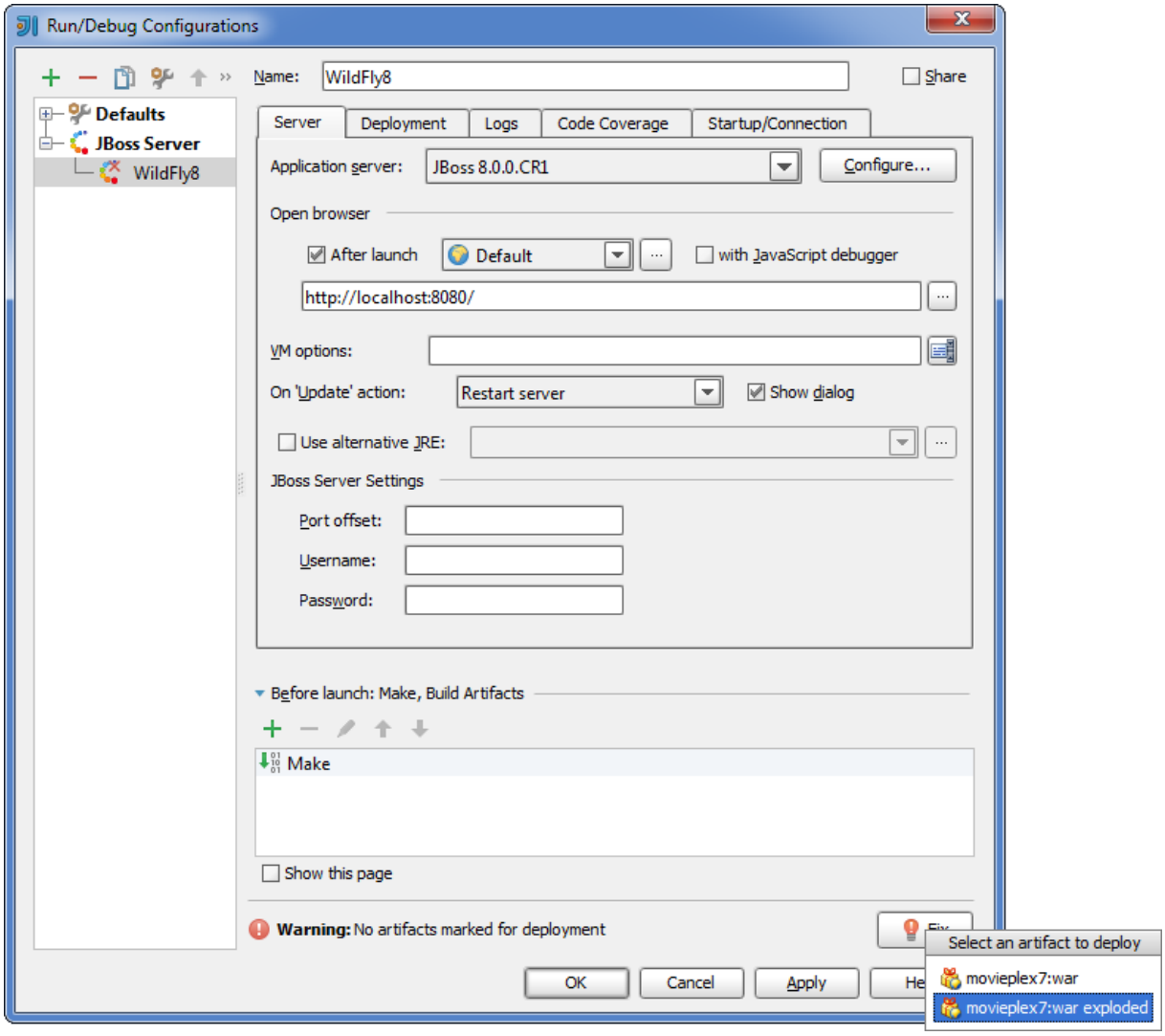
2. In the **Run/Debug Configurations** dialog, click + , select **JBoss Server**, and then select **Local**.



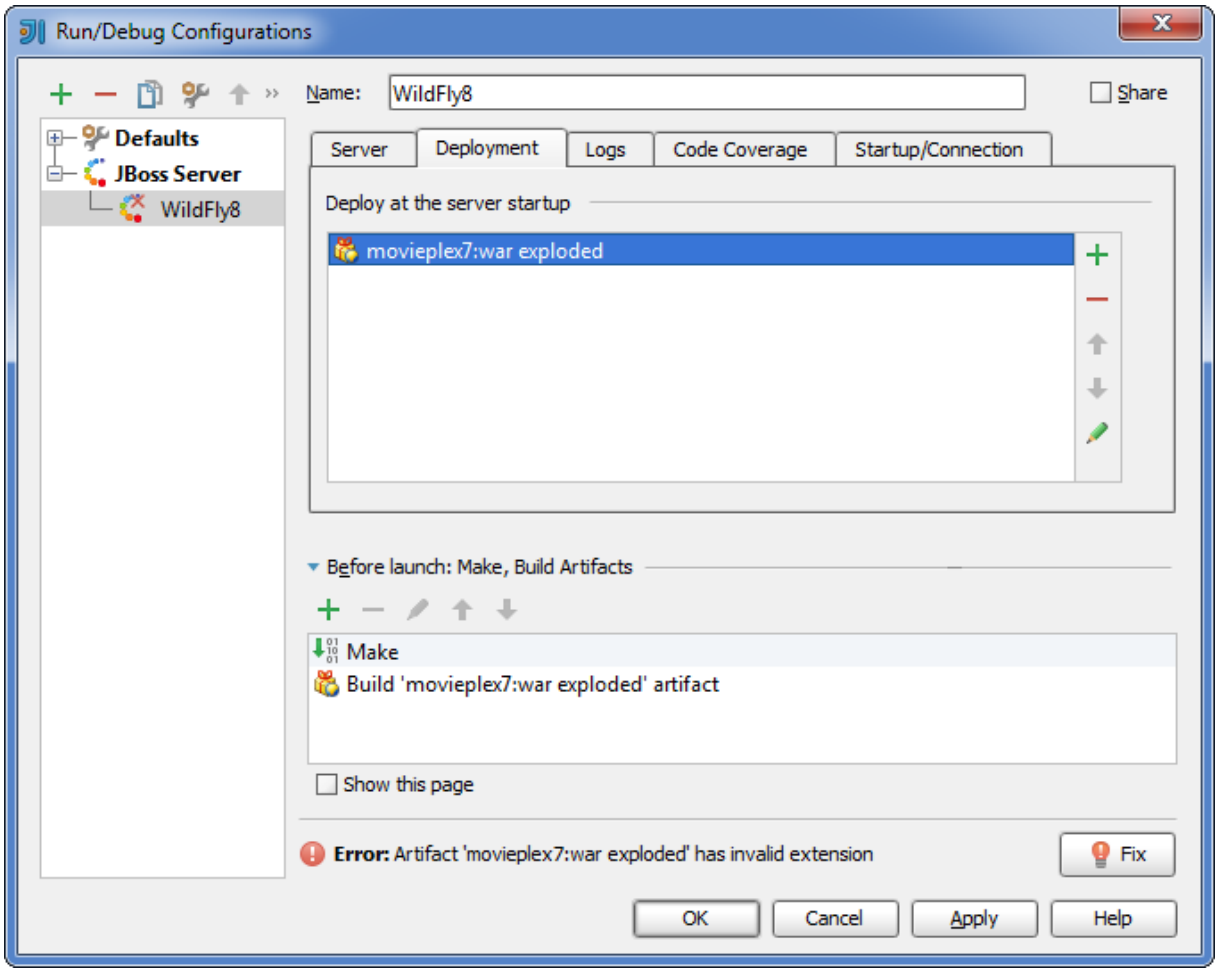
As a result, the run/debug configuration for the WildFly server is created and its settings are shown in the right-hand part of the dialog.

3. Change the name of the run/debug configuration to `WildFly8` (optional).

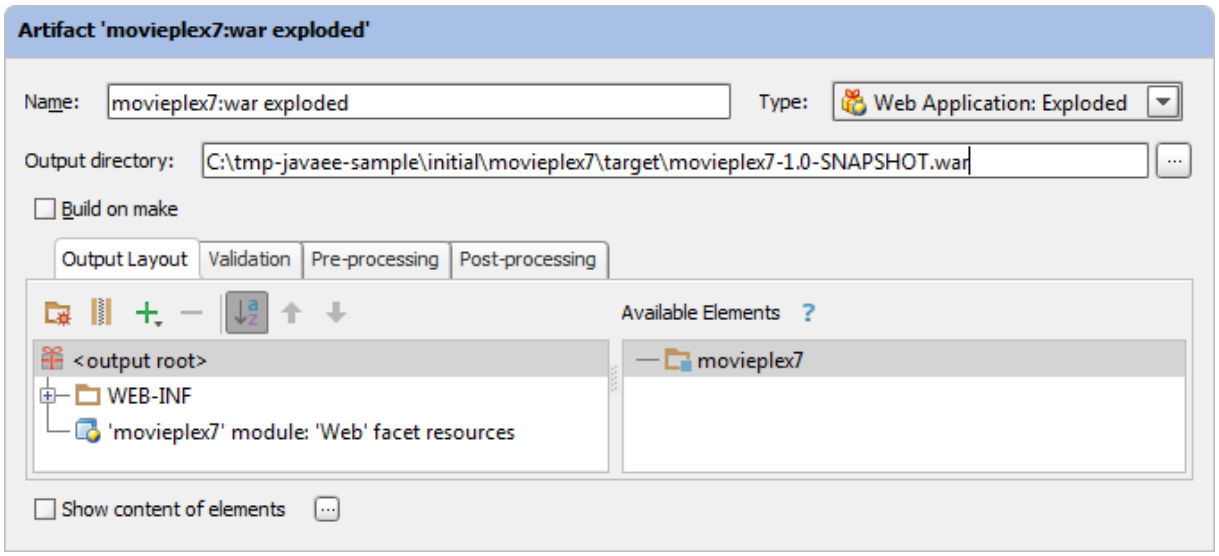
4. In the lower part of the dialog, within the line *Warning: No artifacts marked for deployment*, click **Fix** and select **movieplex7:war exploded**. (Artifacts in IntelliJ IDEA are deployment-ready project outputs and also the configurations according to which such outputs are produced. In our case, there are two configurations for the sample application (*movieplex7:war* and *movieplex7:war exploded*). Both configurations represent a format suitable for deployment onto a Java EE 7-enabled application server. *movieplex7:war* corresponds to a Web archive (WAR). *movieplex7:war exploded* corresponds to the sample application directory structure (a decompressed archive). The second of the formats is more suitable at the development stage because manipulations with it are faster.)



5. Within the line *Error: Artifact 'movieplex7: exploded' has invalid extension*, click **Fix**.

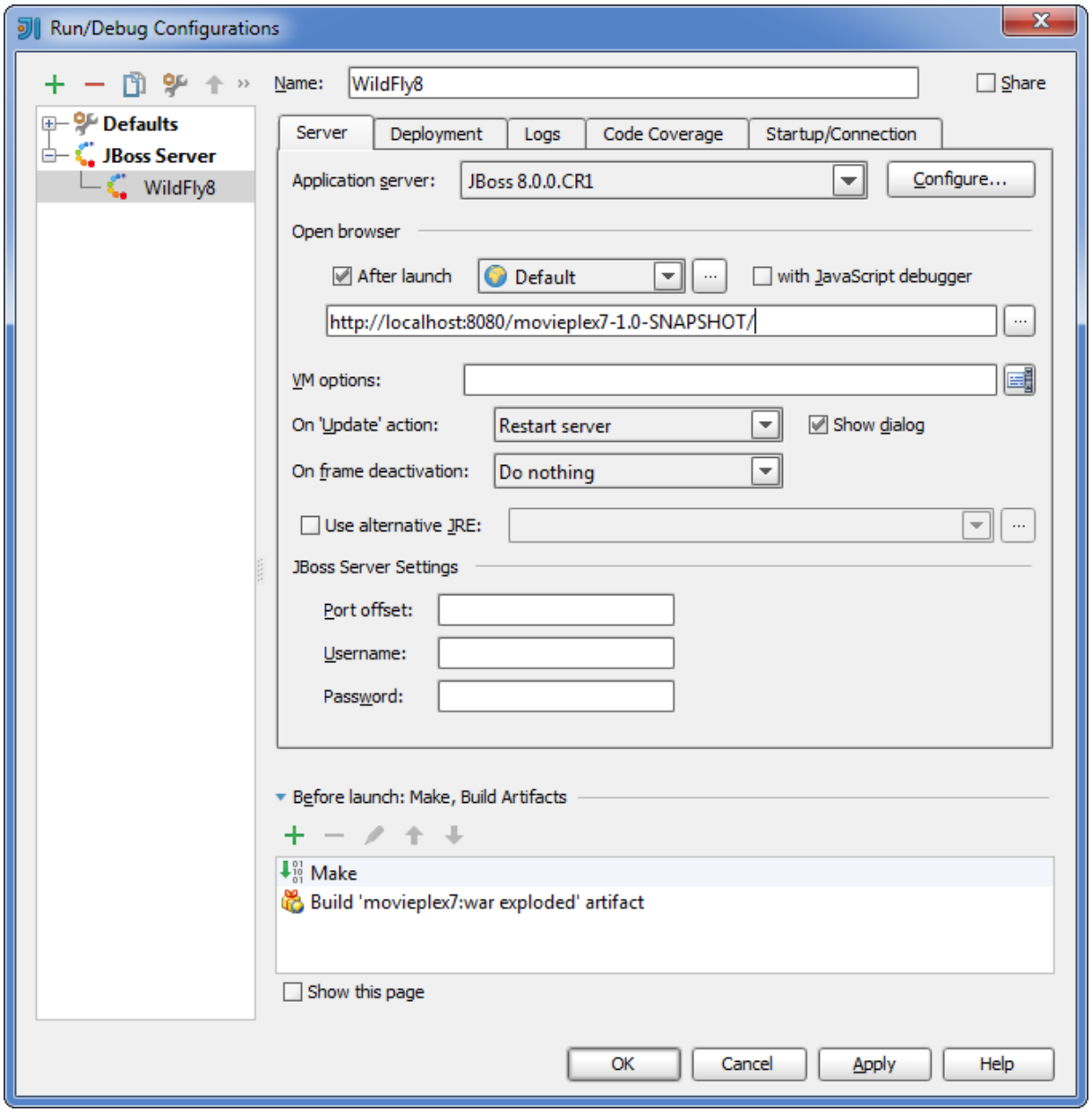


6. In the **Project Structure** dialog, add `.war` at the end of the output directory path, and click **OK**. (For the servers of the JBoss family, the application root directory has to have `.war` at the end.)



7. In the **Run/Debug Configurations** dialog, switch to the **Server** tab. In the field for the application starting page URL,

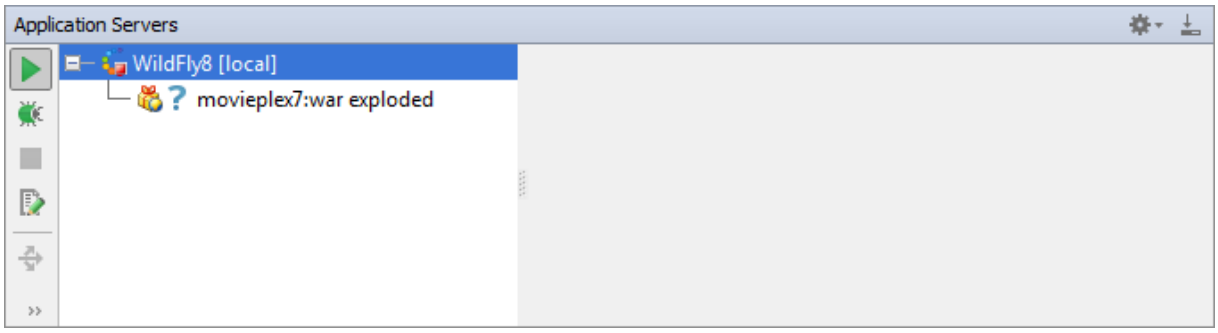
replace `http://localhost:8080/movieplex7-1/` with `http://localhost:8080/movieplex7-1.0-SNAPSHOT/` and click **OK**.



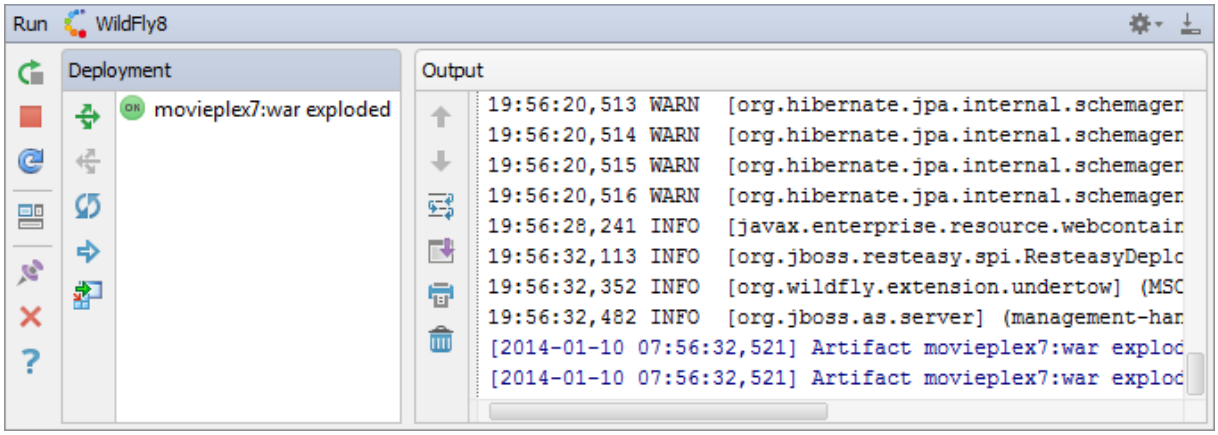
The **Application Servers** tool window opens in the lower part of the workspace. Shown in this window are the server run/debug configuration and the associated deployment artifact. Now you are ready to run the application.

A.2.5. Run the application

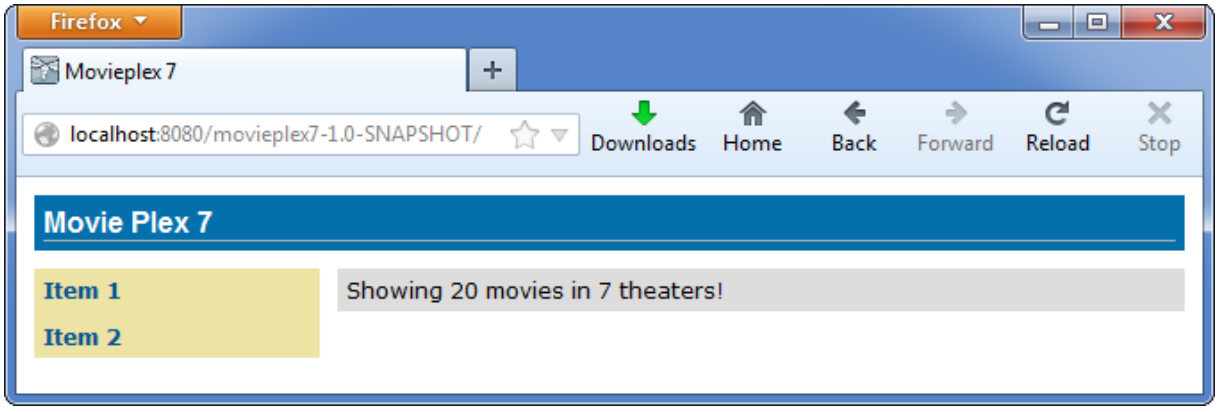
In the **Application Servers** tool window, select the server run/debug configuration (*WildFly8 [local]*) and click **Run**  .



IntelliJ IDEA compiles the code, builds the artifact, starts WildFly and deploys the artifact to the server. You can monitor this process in the **Run** tool window that opens in the lower part of the workspace.



Finally, your default Web browser opens and the starting page of the application is shown.



At this step IntelliJ IDEA is fully prepared for your development work, and you can continue with your exercises.