Developing Web Services Using Java Technology

Activity Guide - Windows

DWS-4050-EE6 Rev A

D65185GC11
Edition 1.1
November 2012
D69080

ORACLE

Disclaimer

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

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

Sun Microsystems, Inc. Disclaimer

This training manual may include references to materials, offerings, or products that were previously offered by Sun Microsystems, Inc. Certain materials, offerings, services, or products may no longer be offered or provided. Oracle and its affiliates cannot be held responsible for any such references should they appear in the text provided.

Restricted Rights Notice

If this documentation is delivered to the U.S. Government or anyone using the documentation on behalf of the U.S. Government, the following notice is applicable:

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

Trademark Notice

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

This page intentionally left blank.

This page intentionally left blank.

Table of Contents

About This Workbook

Lab Goals

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

- Describe how web services frameworks can be used to deploy and consume services.
- Use the JAX-WS technology to deploy and consume web services.
- Use the JAX-RS technology to deploy and consume web services.
- Deploy web services that also leverage other Java™ Platform, Enterprise Edition (JavaEE)technologies, such as the web container infrastructure, Enterprise Java Beans, or the Java Persistence API.
- Apply best practices and design patterns when designing and deploying web services using either JAX-WS or JAX-RS technologies.

Conventions

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

Icons

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

Discussion – Indicates a small-group or class discussion on the current topic is recommended at this time.

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

Caution – Indicates that there is a risk of personal injury from a nonelectrical hazard, or risk of irreversible damage to data, software, or the operating system. A caution indicates that the possibility of a hazard (as opposed to certainty) might happen, depending on the action of the user.

Typographical Conventions

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

Use `ls -al` to list all files.
`system% You have mail.`

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

The `getServletInfo` method gets author information.
The `java.awt.Dialog` class contains `Dialog` constructor.

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

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

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

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

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

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

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

`Courier italic bold` represents variables whose values are to be entered by the student as part of an activity; for example:

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

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

Read Chapter 6 in the *User's Guide*.
These are called *class* options.

Additional Conventions

Java™ programming language examples use the following additional conventions:

- `Courier` is used for the class names, methods, and keywords.
- Method names are not followed with parentheses unless a formal or actual parameter list is shown; for example:

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

  "The `doIt()` method..." refers to a method called `doIt` that takes no arguments.
- Line breaks occur only where there are separations (commas), conjunctions (operators), or white space in the code. Broken code is indented four spaces under the starting code.

Troubleshooting Tips

Below is a list of common error messages you may encounter in this course. An explanation of the problem and a workaround or solution is provided for each.

Performing a Clean and Build Fails with an Error message

Sample Error Message:

```
D:\labs\student\projects\AuctionApp\nbproject\build-
impl.xml:782: Unable to delete file
D:\labs\student\projects\AuctionApp\dist\AuctionApp.jar
BUILD FAILED (total time: 0 seconds)
```

Solution:

The Glassfish server holds a lock on this jar file causing the error. Stopping the Glassfish server before the clean and build eliminates the error. If there are no changes in the included jar, it is ok to ignore the error and just continue with a build.

Web Service Application Stops Working After a Minor Change

After making a minor change to the web service code, model related exceptions are thrown and the application does not run. For example:

```
java.lang.IllegalArgumentException: Object:
Model.Item@af5879 is not a known entity type.
```

Workaround:

NetBeans automatically redeploys the application after each change when working in a Java EE container. This seems to cause an inconsistent state in the persistence environment which causes the exceptions.

To stop the error messages, after completing any changes, restart the Glassfish server and redeploy the application.

Fixing General Errors and Issues not Covered Above

Because of all the generated code in these labs the Glassfish server and/or NetBeans may end up in an inconsistent state. When you receive a random or strange error when you should not, please perform the following steps.

1. Undeploy the application from the GlassFish server.
2. Stop the GlassFish Server
3. Clean and Build your projects
4. Redeploy the application

Cleaning the Auction Database

For each lab, when the application stops running or is undeployed, the tables in the Auction JavaDB database should be deleted. If for some reason the tables cannot be dropped by GlassFish or NetBeans perform the following.

Navigate to the AUCTION database and Tables node in the NetBeans interface. To delete a database, right click it and choose delete. The AUCTION databases must be deleted in the following order.

1. BID
2. AUCTION
3. AUCTIONUSER
4. ITEM
5. SEQUENCE

Lab Setup

Lab Setup and Testing

Objectives

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

- Configure the environment that will be needed for all the exercises in the course.
- Run JUnit tests on the initial model logic for both the Traveller and Auction domains.

Exercise 1: Reviewing the Traveller Project

Task 1 – Reviewing the Use Cases

The Traveller application is the project from which the course draws all of its examples. The Traveller application supports a set of use cases that would necessary to provide an online airline reservation web site, as illustrated by Figure 0-1.

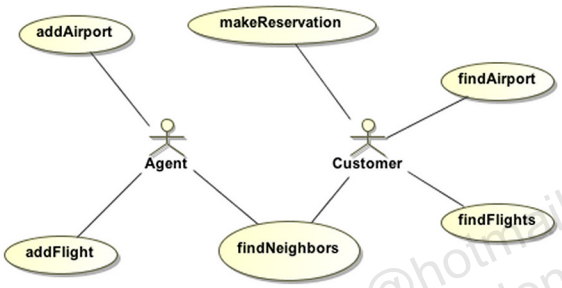

Figure 0-1 Use Cases for the Traveller Project

- `addAirport`

  Adds a new airport to the system.

- `findAirport`

  Finds an airport known to the system, by airport code, or by airport name.

- `findNeighbors`

  Finds all airports that can be reached from a given airport in one flight.

- `findFlights`

  Finds all (direct) flights between two airports.

- `makeReservation`

  Makes a reservation for a customer on a flight. In this system, a round-trip flight requires two reservations, one for each one-way trip.

- `addFlight`

  Adds a flight between the two airports specified to the system.

The uses cases listed above are just a sample of the uses cases that such an application should support - but they are enough to describe all the examples used throughout the course.

Task 2 – Reviewing the Domain Types

Figure 0-2 is a sketch of the domain types used within the Traveller project to capture the information necessary to implement the uses cases described in Figure 0-1. In the design used by the course, all of these types are persistent - and all of these are implements as JPA entity classes, to minimize the effort required to manage their persistence.

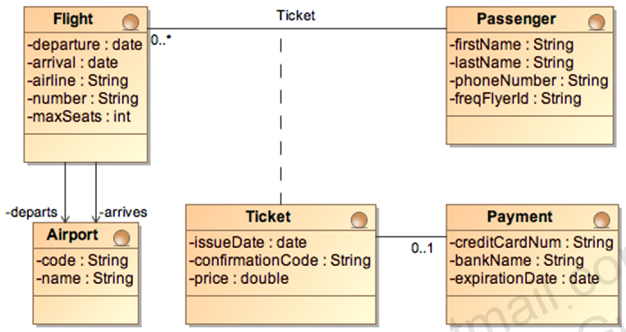

Figure 0-2 Domain Types for the Traveller Project

Task 3 – Reviewing the Data Access Object Types

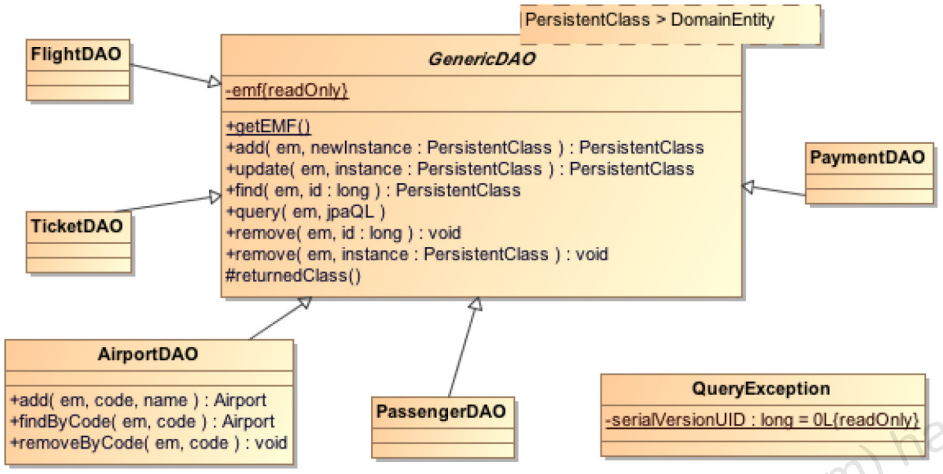

Figure 0-3 DAO Classes for the Traveller Project

Figure 0-3 presents a UML diagram of a number of *Data Access Object* classes, which try to hide the JPA machinery that is used to manage the persistent instances used by the application. In the implementation offered for the examples, these DAO classes will be implemented two ways: as simple Java classes, and as Enterprise Java beans. The first implementation is less successful than the second one at hiding the JPA machinery: absent EJBs, the application logic must manage persistence contexts and transactions explicitly - and this may need to be exposed all the way to business methods.

Exercise 2: Reviewing the Auction Project

Task 1 – Reviewing the Use Cases

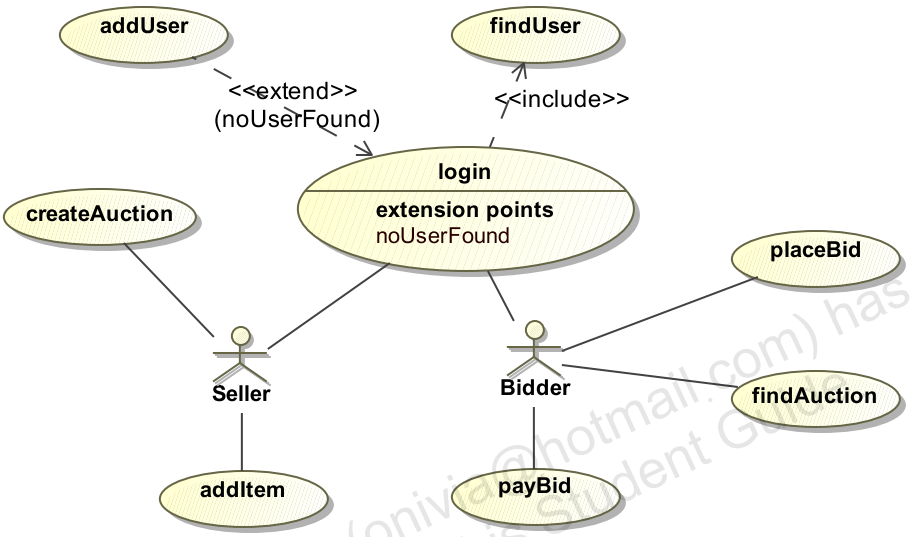

Figure 0-4 Use Cases for Auction System

The Auction application supports the set of use cases necessary to provide an online auction web site, as illustrated by Figure 0-4.

- `login`

  All users of the Auction system must first log on to the system, before they can invoke other operations. To log on, all users must authenticate themselves (perhaps by providing a userid and a password). This use case includes either of the addUser or findUser use cases.

  It might be important to distinguish between sellers and bidders, since there are tasks in the Auction system that only one kind of user may request.

- `addUser`

  Adds a new user to the system database. This use case is invoked by the login use case, when a new user accesses the Auction system for the first time. In addition to the user ID and password already supplied, the system asks the user for additional personal information: name, address (both location and email). Optionally, the user will be able to provide credit card information to be used as a default method of payment.

- `findUser`

Finds an existing user in the system database. This use case is invoked by the login use case, when an existing user returns to the Auction system.

- `addItem`

  Sellers are allowed to create items to be auctioned off. The item to be auctioned and the auction for the item are considered to be independent from each other. This allows a seller who has multiples of a given item to hold separate auctions for each copy, while reusing the same item for each auction. Items hold information, such as the name for the item, a description, and a picture.

- `createAuction`

  Sellers are allowed to create auctions, to auction items off. Each auction holds information, such as the item being auctioned, the initial bid price, the date the auction starts and the date it will end (or how long the auction will last), and a list of all bids placed on it.

  The Auction system should also allow the seller to create multiple auctions for a number of copies of the same item conveniently. This can be achieved by allowing the seller to create a first auction, and then allowing the seller to create "another just like it", or by allowing the seller to specify the number of copies of an auction to create, as part of the process of creating each auction.

- `placeBid`

  Bidders are allowed to bid on auctions. Given an auction, the bidder is allowed to enter a bid on that auction. The Auction system should not allow a bidder to place a bid on an auction that is lower than the current bid price for that auction. The Auction system should also allow the user to find out what the current bid price for the auction is, or even notify the user automatically when the current bid price for the auction changes, or when the bidder is no longer the current high bidder for the auction.

  The bidder should be able to select a collection of auctions they are interested in bidding on, and to allow the bidder to place bids on any of the auctions in that collection. In this case, the Auction system should present the bidder with the current bid price for each of the auctions in the collection, plus a running total of the amount of money the bidder is currently bidding on all auctions in the list, taken together. Also, bidders should be able to find out what the current bid prices for all auctions in the list are, or be notified automatically when the current prices change, or when the bidder is no longer the high bidder for any of the auctions in the collection.

- `findAuction`

  Users must be able to provide criteria to be used to retrieve a collection of bids in which the user might be interested. The user must also be able to narrow down the results of a previous query.

These query operations must be supported while other users are using these same auctions. That is, users must be allowed to find auctions of interest while other users do the same, or bid on some of the same auctions.

- `payBid`

  Bidders must be able to pay for their winning bid on an auction, once the action is over. The Auction system offers to use the credit card information the user entered when the user first registered with the system, although it also allows the user to provide an alternative means of payment. The system should also notify the seller of that auction, when the winning bidder actually pays for that auction.

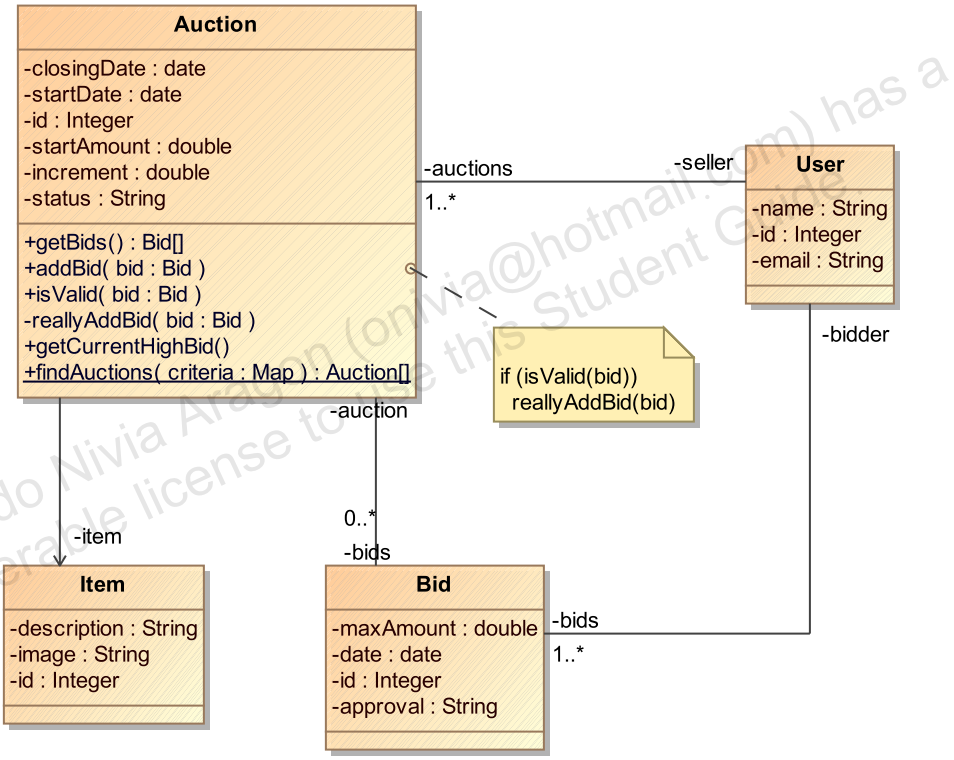

Figure 0-5 Domain Types for Auction System

The business model for the Auction system represents all the information necessary to support the use cases listed previously. Figure 0-5 illustrates the following abstractions, as a starting point to build the business tier for this application:

- `Auction`

  Each instance represents an auction. It is responsible for the dates the auction starts and ends, the initial and winning bid price, and a list of all bids placed on this auction.

- `Item`

  Each instance represents an item that can be auctioned off. It is responsible for information about each item, such as its name, description, or a picture of the item.

- `Bid`

  Each instance represents a bid that a user has placed on an auction. It is responsible for information about the bid, such as the maximum amount the bidder was willing to bid on this bid, the time the bid was placed, and the bidder who placed the bid.

- `User`

  Each instance represents a user of the Auction system. It is responsible for information about each user, such as the user's name, address, email, or credit card information.

Exercise 3: Configuring the Lab Environment

In this exercise you complete the configuration steps required to bring your lab environment to its initial state, prior to the first course lab. During the course, you will develop, deploy, and test services that manipulate persistent objects in two different domains, airline reservations (the `Traveller` application) and online auctions (the `Auction` application). In order to do so, you configure a back-end database which is available to NetBeans, your applications, and the GlassFish application server.

Task 1 - Configuring Java™

This course is written to the match the JAX-WS 2.2, JAXB 2.2, and Metro 2.0 specifications, since they are they ones incorporated in Java™ Platform, Enterprise Edition (JavaEE) 6. Unfortunately, the versions of JAX-WS and JAXB distributed with Java SE 6 Update 17 are different. This can cause problems due to version incompatibilities. In this task you copy two files from the GlassFish installation folder to the Java `endorsed` directory to override the versions of JAXWS and JAXB built into Java 6 Update 17. Complete the following steps.

1. Navigate to `D:\Program Files\Java\jre6\lib`. Create a directory named `endorsed`.

   ```
   $> cd "D:\Program Files\Java\jre6\lib"
   $> md endorsed
   ```

2. Navigate to the new directory.

   ```
   $> cd endorsed
   ```

3. Issue the following two `copy` commands.

   ```
   copy "D:\Program Files\sges-v3\glassfish\modules\endorsed\webservices-api-osgi.jar"
   copy "D:\Program Files\sges-v3\glassfish\modules\endorsed\jaxb-api-osgi.jar"
   ```

Task 2 - Configuring JavaDB Databases

You need two databases for the development projects associated with the course: one for the `Traveller` application used during lectures, and the other for the `Auction` application you develop as you complete the different projects presented during the course.

To create a database, complete the following steps:

1. Open the NetBeans IDE.
2. Select the *Services* tab.
3. Right-click *Databases* > *Java DB*.
4. Select the *Start Server* option.

   The *Output* pane displays the messsage `Apache Derby Network Server - 10.5.3.0 - (802917) started and ready to accept connections on port 1527`.
5. Create a new database.
   a. Right-click the *Java DB* node.
   b. Select *Create Database*.

      A dialog box appears.
   c. Enter **`travel`** for the *Database Name*, *User Name*, and *Password* parameters.
   d. Click *OK*.

      A JDBC connection `jdbc:derby://localhost:1527/travel[travel] on TRAVEL` is added to the *Databases* tree.

Note – You can establish or shut down connections here by right-clicking the connection and then selecting from among the options that appear in the pop-up menu. Once you have established one of these connections, you can also use this node to browse the content of the associated database.

6. Repeat step 5 to create a second database. Enter **`auction`** for the *Database Name*, *User Name*, and *Password* parameters.

   A JDBC connection is added to the *Databases* tree.

Note – There is no need to explicitly create the tables that the applications will use . Both course examples and exercises are built using JPA. The JPA runtime will create the database schema for the persistent classes used by the application automatically.

Task 3 - Configuring JDBC Drivers for Application Use

The examples used throughout the course are available as NetBeans projects in your lab files folder. The list of projects available includes:

- `projects\TravellerEntities`
  This project contains the domain objects listed for the Traveller project. The project contains both a set of domain types implemented as persistent objects using JPA, and a set of data access types (DAOs), to hide the dependency on JPA as much as possible.
- `projects\TravellerEJBs`
  This project reimplements the DAOs listed in the TravellerEntities project so that they are all implemented as session beans. This provides better encapsulation of the JPA machinery within the session beans.
- `examples\jaxrs\ResourceManagers`
  This project contains all the JAXRS examples used in the course that are implemented as POJOs, and deployed within a simple JVM.
- `examples\jaxrs\JavaEEJAXRS`
  This project contains all the JAXRS examples used in the course but implemented as JavaEE components, or deployed to a JavaEE container.
- `examples\jaxws\Managers`
  This project contains all the JAXWS examples used in the course that are implemented as POJOs, and deployed within a simple JVM.
- `examples\jaxws\JavaEEJAXWS`
  This project contains all the JAXWS examples used in the course but implemented as JavaEE components, or deployed to a JavaEE container.
- `projects\AuctionApp`
  This project contains all the domain types listed for the Auction project. The project contains both a set of domain types implemented as persistent objects using JPA, and a set of data access types (DAOs), to hide the dependency on JPA as much as possible.
- `projects\AuctionEJBs`
  This project contains all the domain types listed for the Auction project. The project contains both a set of domain types implemented as persistent objects using JPA, and a set of data access types (DAOs) implemented as stateless session beans, to hide the dependency on JPA as much as possible.

All of the projects you will build or use in this course depend on the presence of a NetBeans library called `DerbyJDBC`, which includes the JDBC drivers required to talk to JavaDB, the persistence engine implemented in Java and distributed with NetBeans and GlassFish. Although these drivers are actually available with GlassFish, NetBeans is missing this library entry. As a result, if you try to open any of the projects distributed with the course, NetBeans will complain about a missing library called `DerbyJDBC`. In this task you add the missing entry for the DerbyJDBC library to NetBeans. Complete the following steps.

1. On the NetBeans main menu, select *Tools* > *Libraries*.
2. Select *New Library*.

   A dialog box is displayed.
3. Enter **`DerbyJDBC`** as the name of the new library.
4. Click *OK* to submit this dialog.

   The original dialog, which lists all libraries, now includes an entry for `DerbyJDBC`.
5. Ensure that `DerbyJDBC` is the selected library. Click *Add Jar/Folder*.

   A dialog box is displayed to allow you to browse for the appropriate file.
6. Select the *JavaDB JDBC* driver jar file, which can be found in `D:\Program Files\sges-v3\javadb\lib\derbyclient.jar`.
7. Click *Add Jar/Folder* to submit this dialog.
8. Click *OK* to close the original *Library Manager* dialog.

Task 4 - Configuring the Ant Classpath

By default, the version of Ant embedded within NetBeans 6.8 will use the versions of JAX-WS and JAXB built into the underlying Java platform. This course was written to match the newer version of these two technologies, as required for JavaEE 6. These are all represented by a jar file called `appserv-rt.jar` within the GlassFish installation directory. In this task you add this jar file to the Ant classpath. Complete the following steps.

1. In the NetBeans menu, select *Tools* > *Options*.

   The NetBeans configuration dialog is displayed.
2. Select the *Miscellaneous* tab, and then select the *Ant* tab.
3. Click *Add JAR/ZIP* to add a new entry to the Ant classpath.
4. Select the file `D:\Program Files\sges-v3\glassfish\lib\appserv-rt.jar`.

5. Click *OK* to add the JAR file to the classpath.
6. Click *OK* again to dismiss the *Options* dialog box.

Exercise 4: JUnit Testing `Traveller` Domain Classes

You can see the domain types that we provide for the Traveller project "in action" by running the JUnit tests that we provide within the TravellerEntities project. These JUnit tests exercise the DAOs provided within the project, which then add and manipulate persistent objects of the types in the project.

To run the JUnit tests for the Traveller project:

1. Click on the `File` menu, then choose `Open Project...`
2. Open the `TravellerEntities` project located in the `D:\labs\student\projects` directory.
3. Navigating to that location shows a dialog like the one in Figure 0-6.

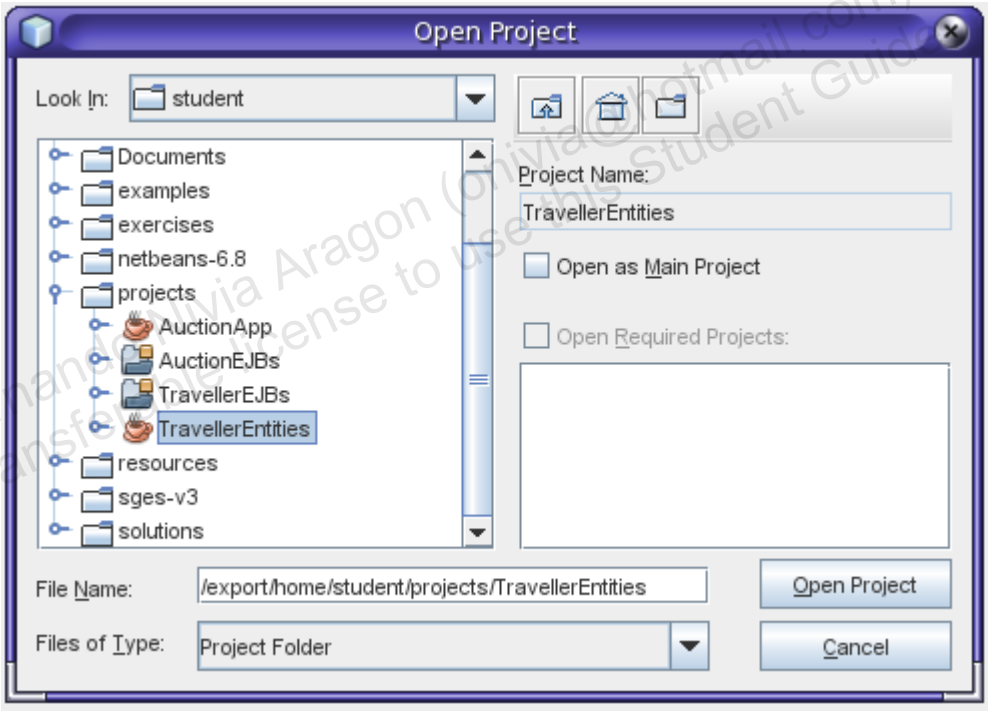

Figure 0-6 Opening Projects

NetBeans Tool Reference Guide: Java Development > Java Application Projects > Opening Projects

- Navigate to the `Test Packages` node in the `Projects` tab, within the `TravellerEntities` project you just opened. Figure 0-7 illustrates what the project structure will look like.

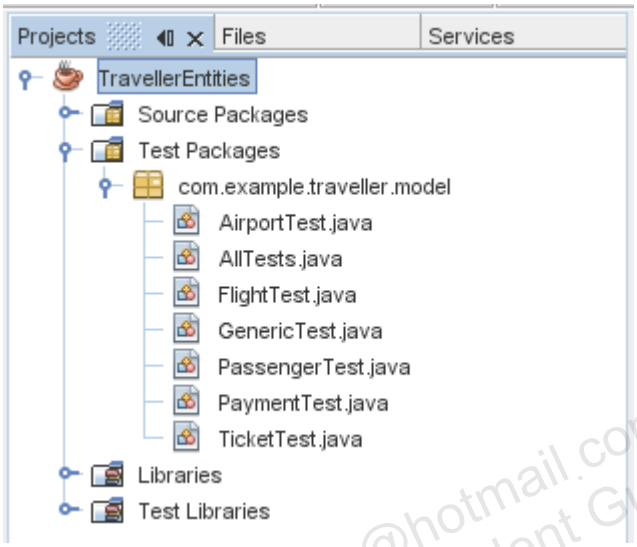

Figure 0-7 `TravellerEntities` Project

Run the unit tests.

1. Open the `Test Packages` node in the project tab.
2. Open the `com.example.traveller.model` package.
3. Right click the `AllTests.java` and choose *Run File*.

NetBeans Tool Reference Guide: Java Development > Java Classes > JUnit Test Classes

Exercise 5: JUnit Testing `Auction` Domain Classes

Test the domain types provided for the Auction project by running the JUnit tests that within the AuctionApp project. These JUnit tests exercise the DAOs provided within the project, which then add and manipulate persistent objects of the types in the project. Run the `ItemTest.java` and `UserTest.java` test classes.

You should leave this project open, after running the JUnit tests, as other projects will require this one.

Exercise Summary

Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercise.

- Experiences
- Interpretations
- Conclusions
- Applications

Exercise Summary

Lab 1

Introduction to Web Services

Objectives

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

- Describe the characteristics of a web service.
- Describe the advantages of developing web services within a JavaEE container.

Exercise 1: Describing Web Services

1. Which of the following are characteristics of a web service? (Choose all that apply)
   a. Tightly-coupled
   b. Loosely-coupled
   c. Interoperable
   d. Platform-specific
2. Indicate for each of the following whether it is a characteristic of traditional RPC solutions or web services. (Write either `RPC` or `Web Services` next to each.)

Local to an enterprise	
Message-driven	
Procedural	
Firewall-friendly	
Variable transports	

3. Mark in each cell in the table whether the web services technology listed exhibits the characteristic listed:

	SOAP	REST
Use XML		
Has Formal Definition		
Relies on HTTP		

4. Fill in the ovals in Figure 1-1 with numbers, indicating the order in which the steps illustrated in this figure are executed:

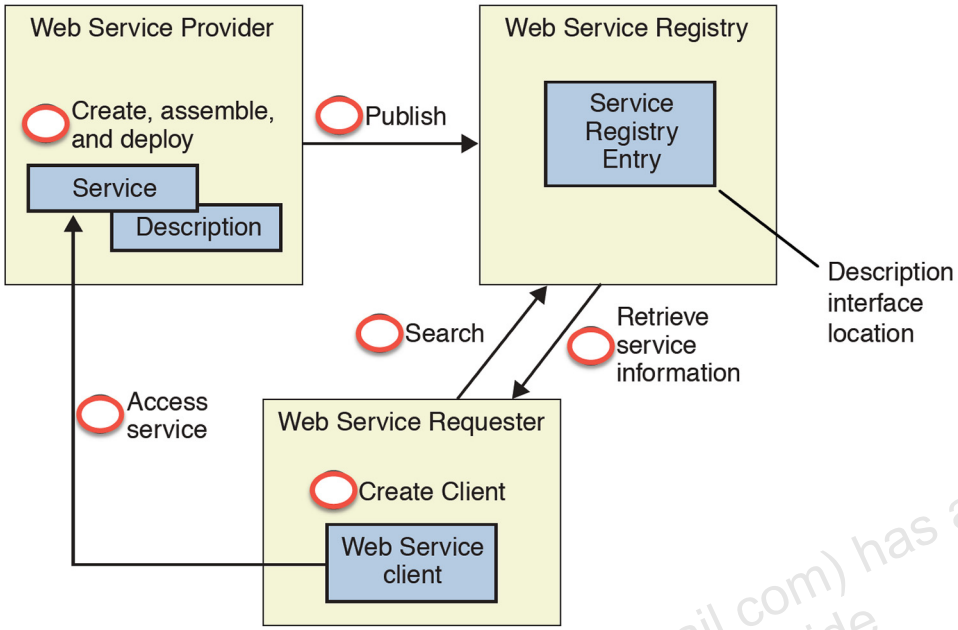

Figure 1-1 Web Services Life Cycle

5. Select the features that correspond to support that the EJB component model offers web services developers. (Choose all that apply.)
   a. Programmatic transaction control.
   b. Declarative transaction control.
   c. Scalability through pooling.
   d. Availability through robustness of Java implementation/runtime.
6. Which of the choices below constitutes deployment choices that are open to the web services deployer? (Choose all that apply.)
   a. Stand-alone Java applications on any JVM, because of web server machinery built-into JAX-WS and JAX-RS runtime.
   b. Servlet endpoints deployed to a web container.
   c. EJB service endpoints.
   d. JMS-based runtimes, because the choice of a JMS transport is required by the web services specifications.
7. Web services and their clients can interact with each other regardless of the platform on which they are running. This is possible in a SOA by means of (Choose one.):
   a. Compatibility

b. Interoperability

c. Accessibility

d. Security

8. Which is the only characteristic of a service that a requesting application needs to know about in SOA? (Choose one.)

a. Private interface

b. Protected interface

c. Public interface

d. Package interface

Exercise Summary

Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercise.

- Experiences
- Interpretations
- Conclusions
- Applications

Lab 2

Using JAX-WS

Objectives

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

- Create simple web services using JAX-WS:
  - Code-first, starting from Java classes
  - Contract-first, starting from WSDL descriptions
- Deploy web service providers using just JavaSE.
- Create and Run web service clients.

Exercise 1: Understanding JAX-WS

1. Identify the correct tool to be used for generating JAX-WS artifacts in each of the following cases:
   a. From a URL of a WSDL file
   b. From a Java implementation source file
   c. From a Java class file
2. Which is the annotation used to describe an argument of a web service operation. (Choose one.)
   a. `@WebResult`
   b. `@WebService`
   c. `@WebMethod`
   d. `@WebParam`
3. Building web services according to the *contract-first* approach is considered a best practice because:
   a. Writing WSDL descriptions first is more maintainable, because WSDL descriptions are more concise.
   b. Writing Java code first is more flexible, because the designer can capture characteristics of the service in Java code that cannot be captured in WSDL.
   c. Writing Java code first is safer, because the developer can capture constraints in Java code that cannot be captured in WSDL.
   d. Writing WSDL descriptions first is safer, because the developer can capture constraints in WSDL descriptions that cannot be captured in Java code.

Exercise 2: Create, Deploy, and Test a JAX-WS Web Service Code-First

In this exercise create, deploy, and test a simple web service for the `Auction` application. Implement this web service using a *code-first* approach. First, implement the Java service provider class. Then, turn the class into a web service through the introduction of JAX-WS annotations. When the web service is ready, deploy it to a stand-alone Java VM, and test it using the SoapUI plug-in for NetBeans.

Task 1 – Define POJO Class

A simple service included in the `Auction` project could be an `ItemManager`. This service allows clients to create, find and manage the set of persistent `Item` descriptions used by the system.

When sellers run auctions on the system, they have to enter a description for the item to be auctioned off. This is where `Item` instances come into play. Instead of just creating a new `Item` description for each new auction started, the system allows users to look for existing Items that match the new item. The seller may reuse an existing `Item` for an auction. This features makes it easier set up auctions of similar items.

The `ItemManager` service then is responsible for creating new `Item` instances, and finding existing `Item` instances. The service may remove instances that are no longer used, but that is a more dangerous service to offer, within the `Auction` system. For convenience, limit your API to accept primitive types (and Strings), and to return the same. For example, the add method could return the new `Item` id, and the method to remove an `Item` could take the id of the `Item` to remove.

1. Create a new Java Application project named `WebServices`. For convenience, place this new project in the `exercises` folder in your home directory. Keep it separate from the `examples` and `project` folders, that contain all the prebuilt code.

   Skip the creation of a `Main` class, when the project is created.

NetBeans Tool Reference Guide: Java Development > Java Application Projects > Creating Projects

2. Add the existing `AuctionApp` project to your newly created project to provide library classes. Open the `AuctionApp` project in the `D:\labs\student\projects` directory. Perform the following steps on your project:

a. Right click on your project and select Properties. The Properties dialog appears.

b. Click on Libraries under Categories.

c. Click on Add Project. Navigate to AuctionApp in project and add it here.

d. Click on Add Library. Add JAX-WS 2.2 to your project.

e. Click on Add Libarary. Add JAXB 2.2. to your project.

f. Click on Add Library. Add Eclipselink(JPA 2.0) to your project.

g. Click on Add Library. Add DerbyJDBC to your project.

h. Click OK.

Once complete, your Libraries dialog should look similar to the following.

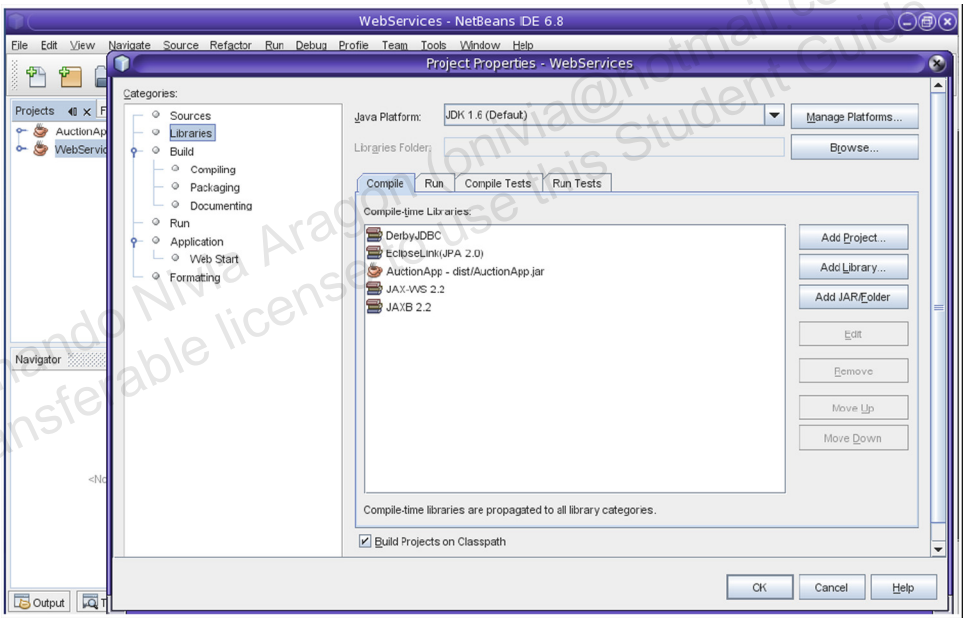

Figure 2-1 Properties for `WebServices` Project

3. Create a new class `ItemManager`, in package `labs`. Define its public API methods with these signatures.

```
public long addItem(String description) { }
public void removeItem(long id) { }
public Item findItem(long id) { }
```

NetBeans Tool Reference Guide: Java Development > Java Classes > Creating Java Classes

4. Add a field to respresent the persistence class.

```
private ItemDAO dao = new ItemDAO();
```

5. Add the following imports to the class.

```
import Model.Item;
import Model.dao.pojo.ItemDAO;
import javax.jws.WebService;
```

Note – As classes are added to your code, fully-qualified class import names are required. NetBeans can fix missing import statements using its *Fix Imports* wizard (available by right-clicking within the source code window), or via "quick fixes" (available by clicking on light-bulbs when they appear on the left margin of the source window).

6. Implement the business logic associated with each of the methods in `ItemManager`.

   All model classes are provided in the `AuctionApp` project. Use `Model.Item` to represent `Item`. To persist data, examine the `Model.dao.pojo.ItemDAO` class and its parent `Model.dao.pojo.GenericDAO`.

   Pass null as the first parameter to `ItemDAO` methods so transactions are automatically managed.

7. Compile `ItemManager.java`. Correct any errors.

Note – NetBeans provides a wizard to create a new Web Service using the *code-first* approach, but only for enterprise projects (a web application or enterprise application). Since this project is not an enterprise project, the wizard is not available.

Task 2 – Incorporate JAX-WS Annotations

Once the `ItemManager` compiles, it is time to turn the class into a web service provider.

1. Incorporate JAX-WS annotations to turn `ItemManager` into a web service. At a minimum, add the `@WebService` annotation to the `ItemManager` implementation class to turn it into a web service provider.
2. Compile `ItemManager.java`. Correct any errors.

Task 3 – Customize Project Build Script

`ItemManager` is now a web service implementation class. Next, generate all the additional artifacts that JAXWS requires in order to actually deploy the web service. This requires additional tasks in the project's ant script, `build.xml`.

The `build.xml` script for your project does not appear in the *Projects* tab for your NetBeans project. To edit the file, click on the *Files* tab. This tab shows all the physical files that belong to a project.

1. Replace the existing `build.xml` file with the following code.

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="WebServices" default="default" basedir=".">
    <description>Builds, tests, and runs the project WebSer-
vices.</description>
    <import file="nbproject/build-impl.xml"/>
<taskdef name="wsgen" classname="com.sun.tools.ws.ant.WsGen"/>
<target name="-post-init" depends="">
  <mkdir dir="generated" />
  <mkdir dir="generated/xml" />
  <mkdir dir="generated/src" />
</target>
<target name="-post-compile" depends="-post-init">
<wsgen
sei="labs.ItemManager"
destdir="build/classes"
resourcedestdir ="generated/xml"
sourcedestdir ="generated/src"
keep="true"
verbose="true"
genwsdl="true"
xendorsed="true"
protocol="soap1.1">
  <classpath>
    <pathelement path="${build.classes.dir}" />
    <pathelement location="${reference.AuctionApp.jar}" />
  </classpath>
</wsgen >
</target >
</project>
```

2. Switch back to the *Projects* tab.
3. Build the project again. After the build completes successfully, click on the *Files* tab. Two new directories are present in your project:

   - `generated/src`

   This directory contains the source code for the Java artifacts generated for your `ItemManager` web service.

   - `generated/xml`

   This directory contains the WSDL/XML artifacts generated for your `ItemManager` web service.

   In addition to these, the compiled versions of all Java artifacts will be placed in the `build` folder, along with all your other compiled classes.
4. Add `generated/src` to the *Projects* tab interface.
   a. Click on the *Projects* tab.
   b. Right click on your project and select *Properties*.
   c. In the *Properties* dialog, select *Sources*. There is only one entry in the Source Package Folders table for `src`.
   d. Add a new entry. Click *Add Folder*.
   e. Select the `generated\src` directory.
   f. Click *Open*. The `generated\src` directory has been added to the *Projects* interface.
   g. Click OK.

   The `generated\src` directory now appears as part of your project.

Task 4 – Deploy Web Service to Stand-alone JVM

Deploy your `ItemManager` web service to a stand-alone JVM.

1. Create a new class, `ItemManagerRunner`, in the `labs` package to deploy your web service. Add a `main()` method to the class.
2. In `main()`, create an instance of your web service implementation class `ItemManager`, and name it `service`.
3. Add an import statement for: `javax.xml.ws.Endpoint`
4. Use the `Endpoint` class to publish the instance of the web service implementation class. Provide a full URL for clients to connect to the web service: protocol, hostname, port number, and the path. For example:

   `http://localhost:8081/ItemManagerService`

5. Ensure that the `JavaDB` database engine is already running.
6. Clean and Build the Application. Correct any errors.
7. Run this application.
8. Verify that the web service is running by opening the URL (`http://localhost:8081/itemManager`) in a browser. A simple web page with information about your web service should be displayed.

Exercise 3: Create and Run a Web Service Client

In this exercise, create a simple web service client to test the `ItemManager` web service created in Exercise 2.

Task 1 – Create an Add Item Client

1. Create a new Java Application NetBeans project named `WSClients` in the `exercises` directory.

   Creating web service clients in a different project from the web service provider is a good idea for a couple of reasons:

   - The *Separation of Concerns* principle suggests that web service providers and their matching web service clients ought to be maintained separately.
   - The supporting artifacts required by the web service provider and the web service client, or the way in which those artifacts are created, may not be the same. Generating both in the same project could be confusing.

   Given a WSDL description of the web service, NetBeans uses JAX-WS to generate all the necessary Java artifacts.

2. Add web services support to your project.

   a. Right click your project.

   b. Select New.

   c. Select Web Services Client.

   d. Check the WSDL URL option to specify the WSDL file.

   e. Enter the `ItemManager` URL:
   `http://localhost:8081/itemManager?wsdl`

   f. Click Finish. All the code needed to create a test web service client is added to your project.

Note – Examine the structure of the `WSClients` project, once the web service client configuration step is done. Note the new nodes NetBeans creates for the `ItemManager` web service.

NetBeans Tool Reference Guide: Web Services > Creating Web Service Clients

3. Make sure that the `ItemManager` service is still up and running.
4. Create a new client to add items to the database.
   a. Create a new Java class named `ItemManagerAddClient` in the `clients` package with a `main` method.
   b. Open up the file to edit. (Add the `main` method if it wasn't automatically created.)
   c. Right click inside the `main` method and choose *Insert Code*.
   d. Select *Call Web Service Operation*. A dialog with information about the `ItemManagerService` is displayed.
   e. Navigate to the `addItem` reference and select it.
   f. Click OK. The code needed to connect to the web service is added. The output should be similar to this:

```
try { // Call Web Service Operation
      labs.ItemManagerService service = new
labs.ItemManagerService();
      labs.ItemManager port = service.getItemManagerPort();
      // TODO initialize WS operation arguments here
      java.lang.String arg0 = "";
      // TODO process result here
      long result = port.addItem(arg0);
      System.out.println("Result = "+result);
    } catch (Exception ex) {
      // TODO handle custom exceptions here
    }
```

   g. `arg0` represents the description for the item.

NetBeans Tool Reference Guide: Java EE Development > Web Services > Creating Web Service Clients > Calling a Web Service Operation

Task 2 – Run Add Item Client

1. Edit `ItemManagerAddClient.java`.
2. Change the value of `arg0` to "Monet Painting".

3. Clean and build the project.
4. Run the `ItemManagerAddClient` class. The output in the console should be similar to the following:

```
result = 1
```

5. The ID of the item added is returned. Remember this number as you will use it in the next task.

Task 3 - Create a Find Item Client

1. Use the steps learned in Task 2, create a *find* web service client named `ItemManagerFindClient` in the `clients` package.
2. Change `arg0` so that you look for the ID you created in task 2.
3. Modify the print statement to provide more information about the item returned. For example.

```
System.out.println("ID = " + result.getId() + " Description
= " + result.getDescription());
```

4. Clean and build the project. Correct any errors.
5. Run the `ItemManagerFindClient` class. You should find the item you created.

Note – You can see if your data was added by opening the *auction* connection under *Databases* in the *Services* tab. An AUCTION database is automatically created and a table corresponding to your data type should exist within the database. Right click your table and select *View Data*. Rows corresponding to the data you added should be present. Note the database connection or any node below the main connection node may need to be refreshed to see the latest data.

Task 4 - Shut Down a Web Service

When the web service application is running, NetBeans displays an output panel that contains the output produced by that application. That same panel includes the controls required to shut down, or restart, that application. Figure 2-2 illustrates that output panel, including those two controls.

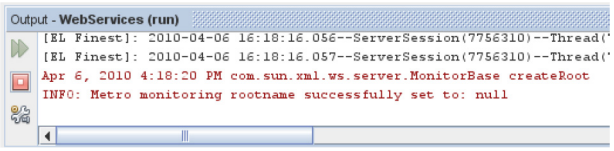

Figure 2-2 Application Output for WebServices Project

In addition to that, NetBeans includes GUI elements that advise the user that there are applications running within NetBeans, and which allow the user to shut them down. These elements appear at the bottom of the NetBeans GUI window, and are illustrated in Figure 2-3.

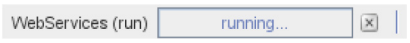

Figure 2-3 Process Indicators

1. Shutdown the `ItemManager` service.

NetBeans Tool Reference Guide: Java Development > Terminating a Running Process

Exercise 4: Create, Deploy, and Test a JAX-WS Web Service Contract-First

In this exercise create, deploy, and test a simple web service for the Auction application. Implement this web service using a *contract-first* approach. First, describe the API the web service implements, using the WSDL vocabulary. Once the WSDL description is ready, implement the Java class that implements the service provider. When the web service is ready, deploy it to a stand-alone Java VM, and test it using web services client.

Task 1 - Define Abstract Description for Web Service in WSDL

The `Auction` system needs a `UserManager` service. A user manager service allows people to register with the system and maintain persistent objects for each registered user. The user management activities the application needs to support include:

- Add new users to the application. This might involve specifying the user's login id, full name, and email address. It could also include a password, to authenticate the user.
- Update information about an existing user, such as their full name, or their email address.
- List all users, or just the users that match some criteria.
- Remove users from the application.

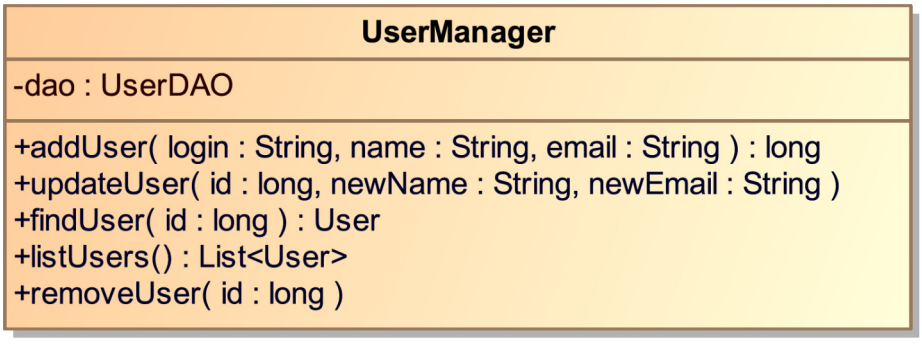

Figure 2-4 Possible UML Diagram for `UserManager` Class

Figure 2-4 shows what a UML diagram for `UserManager` might look like.

Identify the basic methods that such a `UserManager` would need to support, including their mode of interaction, and parameters and return types.

1. For this exercise, work in the `WebServices` project from exercise 2.
2. Write the WSDL description for this web service.

   The WSDL file is a logical description of the web service in terms of the set of operations that this service supports.

   a. To create the file, right click the `labs` package.
   b. Select New.
   c. Select Other.
   d. Select Other.
   e. Select Empty File.
   f. Enter file `UserManager.wsdl` for the file name.
   g. Make sure that the `src\labs` directory is selected.
   h. Click Finish.
   i. Replace the file contents with this WSDL template.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="UserManager.wsdl"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="urn://Auction/"
    targetNamespace="urn://Auction/">

  <types>
    <xsd:schema>
      <xsd:import namespace="urn://Auction/"
          schemaLocation="UserManager.xsd"/>
    </xsd:schema>
  </types>

  <message name="addUserRequest">
    <!-- Your code here -->
  </message>
  <message name="addUserResp">
    <!-- Your code here -->
</message>
  <portType name="UserManager">
<!-- Your code here -->
</portType>
```

```
</definitions>
```

j. Replace the commented sections with your WSDL code for the web service.

3. Create a schema file for your web service in the `labs` package named `UserManager.xsd`.

   An XML Schema file that contains the definitions for all the information carried by any messages between client and web service provider.

   a. Create `UserManager.xsd` using the same steps outlined previously.

   b. Replace the contents of file with the following code.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema id="UserManager.xsd"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="urn://Auction/"
    elementFormDefault="qualified"
    targetNamespace="urn://Auction/">

  <xsd:element name="addUser">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="name" type="xsd:string"/>
        <xsd:element name="email" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="addUserResponse">
    <xsd:complexType>
<!-- Your Code here -->
</xsd:complexType>
  </xsd:element>
</xsd:schema>
```

c. Replace the commented sections with your own code.

4. Create a WSDL file for your web service in the `labs` package named `UserManagerSvc.wsdl`.

   The service WSDL file is the concrete description of the service, which includes the information necessary to actually communicate with the service. This description depends on the logical description defined above.

   a. Create the `UserManagerSvc.wsdl` file as outlined previously.

   b. Replace the contents of the file with the following template.

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

```
<definitions name="UserManagerSvc.wsdl"
    targetNamespace="urn://Auction/"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="urn://Auction/">

  <wsdl:import namespace="urn://Auction/"
      location="UserManager.wsdl"/>

  <binding name="binding" type="tns:UserManager">
    <soap:binding style="document"
      transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="addUser">
      <soap:operation/>
      <!-- Your Code here -->
    </operation>
  </binding>
  <service name="UserManagerSvc">
    <port name="UserManager" binding="tns:binding">
      <!-- Your code here -->
    </port>
  </service>
</definitions>
```

c. Replace the commented sections with your own code.

NetBeans Tool Reference Guide: Java EE Development > Web Services > JAX-WS Web Services > Creating WSDL Files for JAX-WS Web Services

Task 2 – Customize Project Build Script

In the *contract-first* approach, there are more artifacts generated from the WSDL definition, and the actual generation is requested explicitly. In order to write any code in Java that refers to the service described contract-first, the developer needs a Java interfaces for that service. The only way to get an interface is by running the code generator.

1. Replace your build script file `build.xml` with the following build file XML code.

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

```
<project name="WebServices" default="default" basedir=".">
<description>Builds, tests, and runs the project
WebServices.</description>
<import file="nbproject/build-impl.xml"/>

<taskdef name="wsgen"
         classname="com.sun.tools.ws.ant.WsGen"/>
<taskdef name="wsimport"
         classname="com.sun.tools.ws.ant.WsImport"/>
<target name="-post-init" depends="">
  <mkdir dir="generated"/>
  <mkdir dir="generated/xml"/>
  <mkdir dir="generated/src"/>
</target>
<target name="-pre-compile" depends="-post-init">
  <wsimport
    wsdl="${basedir}/${src.dir}/labs/UserManagerSvc.wsdl"
    destdir="build/classes"
    sourcedestdir="generated/src"
    keep="true"
    verbose="true"
    extension="true"
    xendorsed="true"
    package="labs.generated">
  </wsimport>
</target>
</project>
```

2. Build your WebService, to trigger the generation of all Java artifacts associated with the `UserManager` service.

Task 3 - Implement WSDL Web Service

The code generation step produces a Java interface (named after the port type specified in the WSDL description) that describes the API available from the web service. With this information available, implement the web service implementation class.

1. Create a class `UserManagerImpl` in the labs package which implements the `UserManager` interface. See the generated code to determine the methods you must implement.
2. Annotate the class with an `@WebService` annotation.
3. Add the `endpointInterface` attribute to `@WebService` to identify the package location of the generated `UserManager` class.

4. Implement all required methods for UserManagerImpl. Use classes from the `AuctionApp` project to store data. Use the `Model.User` class to represent users. Use the `Model.dao.pojo.UserDAO` class to persist your data.

NetBeans Tool Reference Guide: Java Development > Java Classes > Modifying Java Classes > Overriding Methods

Task 4 - Deploy the Web Service

Once the web service is implemented in Java, there is really very little difference between a *code-first* web service and a *contract-first* one. Go ahead and add the logic necessary to deploy your `UserManager` as a web service. You can use the same strategy that you used for the *code-first* implementation of the `ItemManager` web service that you built before:

1. Create the `UserManagerRunner` class in the labs package. Add a `main()` method to the class.
2. In `main()`, create an instance of your web service implementation class `UserManagerImpl`, and deploy it as a web service.

   Use the `Endpoint` class to publish the instance of the web service implementation class at `http://localhost:8081/userManager`.
3. Ensure that the JavaDB database engine is already running.
4. Clean and build the project. Correct any errors.
5. Run this application. (If NetBeans produces any error messages about the generated code, just dismiss them.)

Note – NetBeans may produce an error stating that the code will not compile. This is an issue with this particular version of NetBeans and will not alter the results, as the code will still compile and run.

Task 5 - Test the Web Service

With the service up and running, create a web service client to test your service. Note that this web service only offers an add service.

1. Create a new project name `UserClient`. Add web service client features to the project as described in Exercise 3.

Note – Make sure the `UserManager` web service is running before adding web service client features to the project.

2. Create a `UserAddClient` class that connects to the `UserManager` web service and adds a user. Once again, use the same techniques you learned in Exercise 3.
3. Run you `UserAddClient` and add a user.

Note – You can see if your data was added by opening the *auction* connection under *Databases* in the *Services* tab. An AUCTION database is automatically created and a table corresponding to your data type should exist within the database. Right click your table and select *View Data*. Rows corresponding to the data you added should be present. Note the database connection or any node below the main connection node may need to be refreshed to see the latest data.

Exercise Summary

Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercise.

- Experiences
- Interpretations
- Conclusions
- Applications

Lab 3

SOAP and WSDL

Objectives

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

- Write WSDL descriptions of web services, using the different styles of SOAP messages that are available.

Exercise 1: Understanding SOAP and WSDL

1. Identify the most popular model and style of SOAP messages from the following list. (Choose one.)
   a. Document literal
   b. Document encoded
   c. RPC literal
   d. RPC encoded
2. State whether the following statements are true or false:
   a. A WSDL descriptor for a web service is required only for a SOAP-based web service.
   b. A WSDL file is embedded within a SOAP message.
   c. WSDL was designed to serve as a standard mechanism for defining the functionality exposed by a web service.
   d. A web service client can retrieve a service descriptor only from the registry.
3. Identify the primary elements of WSDL from the following list. (Choose one.)
   a. Definitions, types, message, portType, binding, service.
   b. Types, portTypes, part, input, and output.
   c. Definitions, part, port, service, and binding.
   d. Definitions, portType, operations, service, and binding.
4. Identify one or more ways by which the WSDL file of a specific web service is located:
   a. `http://eshop.com/CustomerServices/POService?WSDL`
   b. `http://eshop.com/CustomerServices/POService=WSDL`
   c. `http://eshop.com/CustomerServices/POService/WSDL/OrderService.WSDL`
   d. `http://eshop.com/CustomerServices/POService?wsdl=Orderservice.wsdl`
5. State whether the following statements are true or false:
   a. A WSDL descriptor has an abstract model and a concrete model to describe messages and operations.
   b. There are only two types of messaging mechanism; one way and Request/Response.

c. The binding element specifies the data format and the protocol used in the web service. The standard binding extensions are HTTP, SOAP, and MIME.

d. A WSDL file alone cannot be used to generate a web service.

6. Given:

```
<soap:envelope >
    <soap:body >
        <add >
        x xsi:type =" xsd:int">5</x>
        <y xsi:type =" xsd:int">5</y>
        </add >
    </soap:body >
</soap:envelope >
```

What is the style/encoding for this SOAP message? (Choose one.)

a. RPC/encoded

b. RPC/literal

c. Document/literal

d. Document/literal wrapped

7. Which style specifies that each message consist of a single XML node that corresponds to the operation being invoked, with values encoded as simple text? (Choose one.)

a. RPC/encoded

b. RPC/literal

c. Document/literal

d. Document/literal wrapped

Exercise 2: Design a Web Service Contract-First Using the RPC/Literal Style

In Lab 2, you implemented the `UserManager` web service contract-first, by starting with its WSDL description. In this exercise, implement the same contract-first web service, but use the RPC/literal style WSDL description. Many of the techniques learned in Lab 2 are used in this lab.

Task 1 - Setup a new RPC Project

1. Create a new project called `RPCWebServices`.
2. Add the dependent projects and libraries to this project. Below is a quick list of what was added to previous projects.
   - `AuctionApp` project
   - EclipseLink (JPA 2.0)
   - JAX-WS 2.2
   - JAXB 2.2
   - DerbyJDBC
3. Create a Java package called `labs`.

Task 2 - Create a New WSDL Description

The new WSDL descriptions are very similar to those created in Lab 2. Reuse the `UserManager.wsdl` and `UserManagerSvc.wsdl` from Lab 2 and make the changes detailed below.

1. Copy `UserManager.wsdl` and `UserManagerSvc.wsdl` from the last exercise of Lab 2 to the labs directory of this project.
2. Update `UserManager.wsdl` and remove validation.
   a. Delete the following elements from the file

```
<types>
    <xsd:schema>
      <xsd:import namespace="urn://Auction/"
          schemaLocation="UserManager.xsd"/>
    </xsd:schema>
  </types>
```

b. This removes validation from the file.

c. Define web service parameters in this file. Change the first <message> tag to the following

```
<message name="addUserRequest">
    <part name="name" type="xsd:string"/>
    <part name="email" type="xsd:string"/>
  </message>
```

d. Now the addUserRequest elements are defined here rather than in the schema file.

3. Update UserManagerSvc.wsdl to support the RPC style.

a. Find the <soap:binding> tag in the UserManagerSvc.wsdl file.

b. Change the "style" attribute to "rpc" from "document".

c. Find the <soap:address> tag.

d. Change the "location" attribute to "http://localhost:8081/rpcUserManager".

e. Save the file.

Task 3 - Update Project Build Script

Update the project build script to generate web service artifacts.

1. Open the build.xml file for the project.
2. Replace the contents of the file with the following code.

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="RPCWebServices" default="default" basedir=".">
    <description>Builds, tests, and runs the project
RPCWebServices.</description>
    <import file="nbproject/build-impl.xml"/>

<taskdef name="wsimport"
         classname="com.sun.tools.ws.ant.WsImport"/>
<target name="-post-init" depends="">
  <mkdir dir="generated"/>
  <mkdir dir="generated/xml"/>
  <mkdir dir="generated/src"/>
</target>
<target name="-pre-compile" depends="-post-init">
  <wsimport
    wsdl="${basedir}/${src.dir}/labs/UserManagerSvc.wsdl"
```

```
    destdir="build/classes"
    sourcedestdir="generated/src"
    keep="true"
    verbose="true"
    extension="true"
    xendorsed="true"
    package="labs.generated">
  </wsimport>
</target>
</project>
```

3. Save the file.
4. Clean and build the project.
5. As you did before, add the `generated/src` folder to your project sources.

Task 4 - Add Implementation and Runner Classes

Add an implementation and runner class to your project.

1. Copy the `UserManagerImpl.java` and `UserManagerRunner.java` from the last exercise of Lab 2 into the `labs` package of this project.
2. Make one change to the `UserManagerImpl` class. Cast the return value from the `addUser` method as a `long`. For example:

```
return (long) newUser.getId();
```

3. Save the file.
4. Make one change to the `UserManagerRunner` class. Change the published URL for the class to: `http://localhost:8081/rpcUserManager`
5. Save the file.
6. Clean and build the project. Correct any errors.

Task 5 – Deploy Web Service to Stand-alone JVM

In this task, deploy the web service.

1. Run the `UserManagerRunner` class.
2. Verify that the web service is running by opening the URL (`http://localhost:8081/rpcUserManager`) in a browser. A simple web page with information about your web service should be displayed.

Exercise 3: Create a Test Web Service Client

In this exercise, create and test a simple web service client for the new contract-first `UserManager` web service defined above.

Task 1 - Define a Client Class

1. Create a new Java Application project named `RPCClients`.
2. Add support for web service clients to your project.
3. Create the `UserManagerRPCClient` class in the clients package.
4. Use NetBeans to generate the code to connect your client to the web service.
5. Modify the name and email values in your code so test data is added to the database.
6. Clean and build your project. Correct any errors.
7. Run the `UserManagerRPCClient` class and add one user entry.

Note – You can see if your data was added by opening the *auction* connection under *Databases* in the *Services* tab. An AUCTION database is automatically created and a table corresponding to your data type should exist within the database. Right click your table and select *View Data*. Rows corresponding to the data you added should be present. Note the database connection or any node below the main connection node may need to be refreshed to see the latest data.

Exercise Summary

Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercise.

- Experiences
- Interpretations
- Conclusions
- Applications

Lab 4

JAX-WS and JavaEE

Objectives

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

- Deploy POJO web services to a web container.
- Secure POJO web services to a web container, delegating authentication to the container.
- Define a web service in terms of an Enterprise Java Bean.
- Deploy an EJB web service to an EJB container.

Exercise 1: Understanding JAX-WS in Containers

1. Which are considered advantages of deploying POJO web services to a web container? (Choose all that apply.)
   a. HTTP session management built in.
   b. Scalability and availability features built-in.
   c. Declarative transaction management.
   d. Pooling of web service provider instances built in.
2. Which mechanisms are available to define the set of roles that will be available for programmatic authorization within a web service provider class? (Choose all that apply.)
   a. Initialization properties in `web.xml` element corresponding to web service implementation class.
   b. `@DeclareRoles` annotation on web service implementation class.
   c. `@RolesAllowed` annotation on web service implementation class.
   d. security-constraint element in `web.xml`.
   e. `@DeclareRoles` annotation on web service client implementation class.
   f. `security-role-mapping` elements in `sun-web.xml`.
3. What mechanisms are available for a JAX-WS web service deployed to a web container to obtain the identity of the caller? (Choose all that apply.)
   a. `ServletRequest` as a parameter to the web service method call.
   b. `ServletRequest` as a resource injected directly in the web service implementation class.
   c. `SecurityContext` as a resource injected directly in the web service implementation class.
   d. `ServletRequest` obtained from a `MessageContext`.
   e. `WebServiceContext` as a resource injected directly in the web service implementation class.
4. Choose the option that best describes how most of the information about a request that is made available to the web service implementation class is provided. (Choose one.)
   a. Directly via dependency injection.
   b. As properties provided by `WebServiceContext` directly.
   c. As properties provided by `MessageContext` directly.

d. As parameters to the web service method call.

Exercise 2: Deploy POJO Web Services to a Web Container

In Lab 2, you built two simple web services, `ItemManager` (implemented code-first) and `UserManager` (implemented contract-first). Both simple POJO web services are deployed stand-alone on a JVM.

In this exercise deploy both web services in a web application. This way, the functionality provided by the web container provides additional features like a web service creation wizard and an automated test harness.

POJO web services can be deployed either stand-alone, using the HTTP server built into Java SE 6, or to any web container. The implementation of the POJO web service itself does not change based on its deployment environment. Any differences are handled by the JAX-WS runtime.

Task 1 - Create a new Web Application Project

1. Create a new web application project named `POJOsInContainer`.
   a. From the menu select *File* then *New*.
   b. Select *Java Web* then *Web Application*.
   c. Enter the project name and put the project in the `exercises` directory.
   d. Click *Next*.
   e. Keep the default values and click *Finish*.
2. Add the dependent projects and libraries to this project. Below is a quick list of what needs to be added. Note that both JAX-WS 2.2 and JAXB 2.2 are automatically included as part of a web application project.
   - `AuctionApp` project
   - EclipseLink (JPA 2.0)
   - DerbyJDBC
3. Create a Java package called `labs`.

Task 2 - Add Database Features to the Project

In this task, add the database resource for the project.

1. Create a database connection pool named `auctionDbPool`.

   a. Right click the project.

   b. Select *New* then *Other*.

   c. Select the *Glassfish* category.

   d. Select *JDBC Connection Pool*.

   e. Click *Next*.

   f. Name the pool `auctionDbPool`.

   g. Link the pool to the `auction` database connection. (example in Figure 4-1)

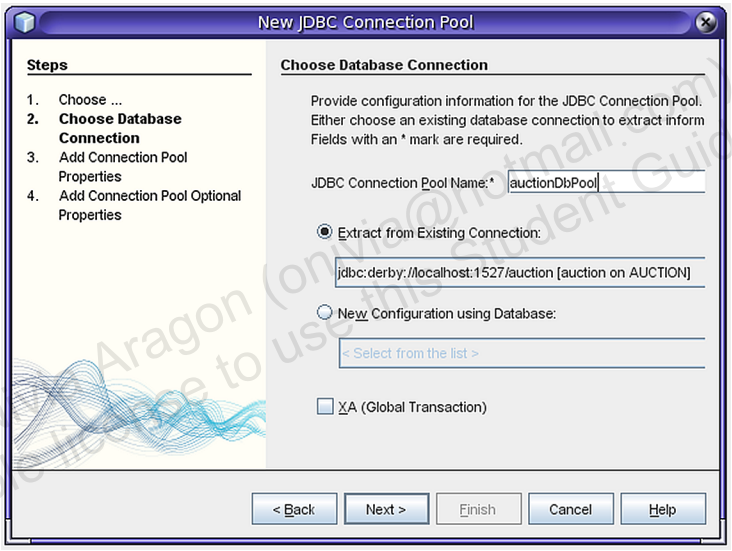

Figure 4-1 Entering New Connection Pool Name

   h. Click `Next`

i. Accept the default values and click *Finish*. (Example in Figure 4-2)

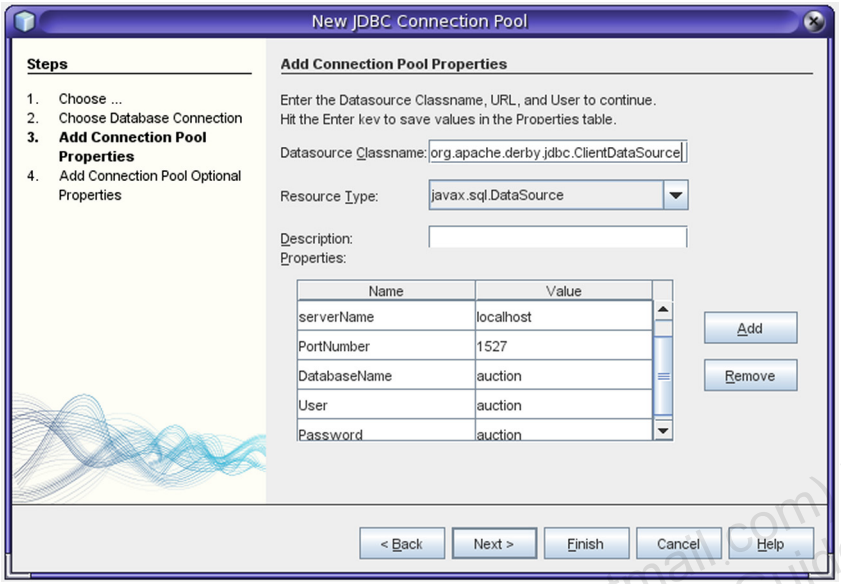

Figure 4-2 Entering JDBC Information for new Connection Pool

2. Configure a JDBC resource to let the application access the database through the `auctionDbPool` via JNDI lookup.
   a. Right click the project.
   b. Select *New* then *Other*.
   c. Select the *Glassfish* category.
   d. Select *JDBC Resource*.
   e. Click *Next*.
   f. Select `auctionDbPool` as your connection pool.
   g. Change the JNDI name to `jdbc/auctionDbPool`.
   h. Click *Finish*.

Task 3 - Create the Code First Item Web Service

In this task, implement the `ItemManger` web service in a Java EE web container.

1. Right click the project.
2. Select *New*.
3. Select *Other*.

4. Select the *Web Services* category.
5. Select *Web Service*.
6. Click *Next*.
7. Enter `ItemManager` for the web service name.
8. Enter `labs` for the package name.
9. Check *Create Web Service from Scratch*.
10. Click *Finish*.
11. Web service code is added to your project. (Mostly XML configuration files in your `nbproject` directory.) Also, `ItemManager.java` is created in the *labs* package. The file contains basic template information.
12. Copy the file contents of `ItemManager.java` from Lab 2, exercise 2 to the clipboard. Replace the contents of `ItemManager.java` with the old code.
13. Replace the @WebService annotation with the following code:

```
@WebService(serviceName="ItemManagerWS")
```

    The serviceName attribute specifies the location of the web service in the context of the project. For example, the new `ItemManager` web service will be available at:
    `http://localhost:8080/POJOsInContainer/ItemManagerWS`
14. Save the file.
15. Clean and build the project. Correct any errors.
16. Run the project and start the web service.

NetBeans Tool Reference Guide: Java EE Development > Web Services > JAX-WS Web Services > Creating an Empty JAX-WS Web Service

Task 4 - Test the Web Service

In this task, test the web service from its Java EE container.

1. Open the web service URL at:
   `http://localhost:8080/POJOsInContainer/ItemManagerWS`

   This should display a simple web page with a link to the generated WSDL file. Just like the web services created before.

2. Glassfish provides a build in test harness for web services. To use this too, open the following URL:

```
http://localhost:8080/POJOsInContainer/ItemManagerWS?Tester
```

3. This page allows you to perform the add, find, and remove operations.
4. Add a new item using the test harness.
5. Find the item you added.
6. View the database to see if the item is present there.

Task 5 - Create a Contract First User Web Service

In this task, create a contract first web service.

1. Copy the `UserManager.wsdl`, `UserManager.xsd`, and `UserManagerSvc.wsdl` from Lab 2, Exercise 4 to the `labs` package.
2. Create the User Manager web service using the NetBean web service wizard.
   a. Right click the project.
   b. Select *New*.
   c. Select *Other*.
   d. Select the *Web Services* category.
   e. Select *Web Service from WSDL*.
   f. Click *Next*.
   g. Enter `UserManagerImpl` for the web service name.
   h. Enter `labs` for the package name.
   i. Click the browse button and navigate to the labs package. Select `UserManagerSvc.wsdl` as the WSDL file for building your web service.
   j. Click *Finish*.
3. The Web Service is created for you automatically by NetBeans. A `UserManagerImpl.java` file is placed in the labs package.
4. Implement the code for the `addUser` method using the code from Lab 2, Exercise 4. Add any required imports.

   Now the web service is ready to deploy. The NetBeans wizard takes care of any other implementation details.
5. Clean and build the project. Correct any errors.

6. Deploy the project.

Task 6 - Test the Web Service

In this task, test the web service from its Java EE container.

1. Open the web service URL at:
   `http://localhost:8080/POJOsInContainer/UserManagerSvc`

   This should display a simple web page with a link to the generated WSDL file. Just like the web services created before.
2. Test the web service using the Glassfish test harness at:

`http://localhost:8080/POJOsInContainer/UserManagerSvc?Tester`

3. Add a new item using the test harness.
4. View the database to see if the item is present there.

NetBeans Tool Reference Guide: Java EE Development > Web Services > JAX-WS Web Services > Creating a JAX-WS Web Service From a WSDL File

NetBeans Tool Reference Guide: Java EE Development > Enterprise Application Projects > Deploying Java EE Applications

Exercise 3: Secure POJO Web Services in Web Container

The two web services `ItemManager` and `UserManager` are deployed to a web container, as part of a web application. One of the advantages of this approach is it is possible to secure the web services, delegating authentication to the web container. In this exercise, secure access to the `UserManager`, so that only authenticated admins and users can create new users.

Task 1 - Create User Accounts

To set up security for the User Manager web service a couple of user accounts are required. Set up two user accounts on your Glassfish server using the administration console.

1. Start the Glassfish server if it is not already running.
2. Open a Web Browser.
3. Open the *Glassfish Adminisration Console* at: `http://localhost:4848`.
4. When prompted for User Name and Password, enter *admin* and *adminadmin* respectively.
5. Navigate the menu tree to *Configuration -> Security -> Realms*.
6. Click on the *File* realm.
7. Click on the *Manage Users* button.
8. Add two users, `tracy` and `kelly`. Set their passwords to “password”. They are not part of any group and their names are case sensitive.
9. Click the *Back* button. This completes the setup of users for this lab.

NetBeans Tool Reference Guide: Server Resources > Java EE Application Servers > Administering Security > Adding a File Realm User

Task 2 - Define Security Constraints on Web Application

The `UserManager` web service defines two kinds of users, administrators and everyone else. When the application is complete, only administrators may add new users to the application. In this task, define the security constraints for the application using XML configuration files.

1. Navigate to the location of the web application configuration files.
   a. Open your project.
   b. Open the Web Pages folder.
   c. Open the WEB-INF folder.
   d. There should be two configuration files there: `sun-web.xml` and `web.xml`.
   e. The `web.xml` file is not created by default. If it does not exist do the following to create it.
      1. Right click the project.
      2. Select *New*, then *Other*.
      3. Select *Web* from categories.
      4. In the right pane, scroll all the way down to the bottom and select *Standard Deployment Description (web.xml)*.
      5. Click *Next*.
      6. Click *Finish*.
      7. The `web.xml` is created for you.
2. Configure the web container to use Basic HTTP authentication.
   a. Edit the `web.xml` file in XML mode.
   b. Add a `<login-config>` element after the `</session-config>` tag. Use the following XML code.

```
<login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>file</realm-name>
</login-config>
```

      This configures Glassfish to use Basic HTTP authentication. Clients must provide a userid and password to perform web service operations. The realm is a match for the user accounts created in Task 1.

   c. Save the file.
3. Add `security-role` elements for the `user` and `administrator` roles.
   a. Edit the `web.xml` file in XML mode.
   b. After the `</login-config>` tag just added to the file, add a `<security-role>` tag for the `user` role using the following XML code:

```
<security-role>
        <description/>
        <role-name>user</role-name>
</security-role>
```

c. Add a security role for `administrator`.

d. Save the file.

4. Setup a security constraint to specify which roles can access the web service URLs.

   a. Edit the `web.xml` file in XML mode.

   b. Add the following code to the file after the `</security-role>` tag. This creates the `AuctionServices` security constraint.

```
<security-constraint>
        <display-name>AuctionServices</display-name>
        <web-resource-collection>
            <web-resource-name>AuctionServices</web-
resource-name>
            <description/>
            <url-pattern><!-- code here --></url-pattern>
            <url-pattern>/<!-- code here --></url-pattern>
            <http-method><!-- code here --></http-method>
        </web-resource-collection>
        <auth-constraint>
            <description/>
            <role-name><!-- code here --></role-name>
            <role-name><!-- code here --></role-name>
        </auth-constraint>
</security-constraint>
```

   c. Replace the commented portions of the XML with the following information.

      1. The URL patterns for the `UserManager` and `ItemManager` web services. This specifies what is to be protected.
      2. Use the `POST` method to submit login information.
      3. Add the `administrator` and `user` roles. The roles specify who can access the URLs above.

   d. Save the file.

5. Link roles to users. Link `tracy` and `kelly` to *administrator* and `user`.

   a. Edit `sun-web.xml` in XML mode.

   b. Add the following code to link `tracy` and `kelly` to their roles in the application. Add your code after the `<context-root>` element.

```
<security-role-mapping>
    <role-name>administrator</role-name>
    <principal-name>tracy</principal-name>
  </security-role-mapping>
  <security-role-mapping>
    <role-name>user</role-name>
    <principal-name>kelly</principal-name>
  </security-role-mapping>
```

c. Save the file.

6. Add code to log successful calls to the `addUser` method.

a. Edit the `UserManagerImpl.java` file.

b. Add the following imports.

```
import javax.annotation.Resource;
import javax.servlet.ServletContext;
import javax.xml.ws.WebServiceContext;
import javax.xml.ws.handler.MessageContext;
```

c. Add the following code to the `addUser` method

```
String caller = context.getUserPrincipal().getName();
    MessageContext msgCtx = context.getMessageContext();
    ServletContext appCtx =
         (ServletContext)
msgCtx.get(MessageContext.SERVLET_CONTEXT);
    appCtx.log("add: caller: " + caller);
```

d. Add the the `context` field to the class.

```
@Resource WebServiceContext context;
```

e. Save the file.

Note – The set of roles that can be specified in the web application's `sun-web.xml` is application-specific. Different web applications deployed to the same web container could each have different roles defined for each application.

Note – For convenience, `sun-web.xml` can also map system-wide roles to application-specific ones. This would allow for the administration of a number of web applications, without requiring a complete of all authorized used separately for each individual application: the system wide assignment of system roles to principals would determine their application-specific role assignments within each web application.

Note – Rather than modify XML configuration files, security constraints can be defined using annotations. Roles are defined by annotating the web service implementation class with an `@DeclareRoles` annotation. An `@RolesAllowed` annotation at class level, specifies default role requirements for calls to any web service operation. An `@RolesAllowed` annotation at the method level, restricts access to that operation. Examples of these annotations and their use is included in the EJB exercises that follow.

Task 3 - Deploy Service in Web Application

In this task, deploy the web application.

In this task, test the web service from its Java EE container.

1. Build the web application. Correct any errors.
2. Deploy the application.
3. Open the web service URL at:
   `http://localhost:8080/POJOsInContainer/UserManagerSvc`

   This should display a simple web page with a link to the generated WSDL file. Just like the web services created before.

   To test the secured web services, in the next task, write a JAX-WS client application that can authenticate when it calls the web service.

Exercise 4: Creating an Authenticating Client

In this exercise, create and test a simple authenticating web service to the secured `UserManager` web service defined above.

Task 1 - Define a Client Class

1. Create a new Java Application project named `SecuredClients`.

Add support for web service clients to your project. The WSDL for the new web service is located at:
`http://localhost:8080/POJOsInContainer/UserManagerSvc?WSDL`

2. Create the `UserManagerSecuredClient` class in the `clients` package.
3. Use NetBeans to generate the code to connect your client to the web service.
4. Save the `UserManagerSecuredClient.java` file.

Task 2 - Modify the Class to Support Authentication

1. Edit the `UserManagerSecuredClient.java` file.
2. Add the following imports.

```
import java.util.Map;
import javax.xml.ws.BindingProvider;
```

3. Add the following code to provide authentication data to the web service.

```
Map<String,Object> reqCtx =
          ((BindingProvider)port).getRequestContext();
reqCtx.put(BindingProvider.USERNAME_PROPERTY, "tracy");
reqCtx.put(BindingProvider.PASSWORD_PROPERTY,"password");
```

4. Save the file.

Task 3 - Test the Client and Web Service

1. Modify the name and email values in your code so test data is added to the database.
2. Build your project. Correct any errors.

3. Run the `UserManagerSecuredClient` class and add one user entry.
4. Try using `kelly` to authenticate. Is the user still added?
5. Try using `bubba` or `hacker` to authenticate. What happens?

Task 4 - Shut Down Web Services

Undeploy the web service application you deployed in this exercise, and shut down the application server.

Exercise 5: Define and Deploy EJB-based Web Services

The web services built so far use data access objects (DAOs) to manipulate persistent domain objects. Once a task becomes complex enough, the web services themselves need to manage the transactions performing the required work by each operation. This could be complex logic to write and maintain.

In this exercise, modify the web services project to implement as Enterprise Java Beans (EJBs). EJBs allow you to delegate the implementation of transaction management to the EJB container. However, transaction semantics can still be specified declaratively.

Task 1 – Copy Existing Web Services to Enterprise Application

In JavaEE6, an enterprise application can be packaged up and deployed as a simple web application. An EJB container and many of the services that enterprise applications rely on are available alongside the JavaEE6 web container. In this lab, take advantage of this new feature.

1. Copy your existing `POJOsInContainer` web application project into a new project, called `EJBWebServices`.
   a. Right-click the project you want to copy.
   b. Choose *Copy*. A dialog will ask you for the destination folder and name for the new project.
   c. Place the project in the same folder as `POJOsInContainer`, and name it `EJBWebServices`.
2. Update the dependent projects and libraries for this project.
   a. Right click the project.
   b. Choose *Properties*.
   c. Select *Libraries*.
   d. Remove the `AuctionApp` project and *all other librabries* still included here from the list of libaries.
   e. Add the `AuctionEJBs` project to the list of libraries. It is located in the same directory as `AuctionApp`.

Note – Until the web service implementation classes are modified in the next task, NetBeans reports problems with those classes. The errors are to be expected and will be fixed in the next task.

Task 2 – Rewrite Existing Web Services as EJBs

In this task, modify the web service implementation classes so that they are implemented as EJBs.

1. Update `ItemManager.java` for EJBs.
   a. Edit `ItemManager.java`.
   b. Replace the POJO DAO import with the following:

```
import Model.dao.ejbs.ItemDAO;
```

   c. Add the following new imports.

```
import javax.annotation.security.DeclareRoles;
import javax.annotation.security.RolesAllowed;
import javax.ejb.EJB;
import javax.ejb.Stateless;
```

   d. Add the following annotations to the `ItemManager` class definition.

```
@DeclareRoles({"user","administrator"})
@RolesAllowed({"user","administrator"})
@Stateless
```

   e. Remove the `null` parameter from all the methods calls in the class.
   f. Change the definition for the `dao` field to:

```
@EJB private ItemDAO dao;
```

   g. Save the file.
2. Update `UserManagerImpl.java` for EJBs.
   a. Edit `UserManagerImpl.java`.
   b. Replace the POJO DAO import with the following:

```
import Model.dao.ejbs.UserDAO;
```

   c. Add the following new imports.

```
import javax.annotation.security.DeclareRoles;
```

```
import javax.annotation.security.RolesAllowed;
import javax.ejb.EJB;
import javax.ejb.Stateless;
```

d. Add the following annotations to the `UserManagerImpl` class definition.

```
@DeclareRoles({"user","administrator"})
@RolesAllowed({"user","administrator"})
@Stateless
```

e. Remove the `null` parameter from all the methods calls in the class.

f. Change the definition for the `dao` field to:

```
@EJB private UserDAO dao;
```

g. Add the `@RolesAllowed` annotation to the `addUser` method. After the change is made, only `tracy` should be able to add users with the web service.

```
@RolesAllowed({"administrator"})
```

h. Save the file.

3. Clean and build the project.
4. Deploy the `EJBWebServices` web application.

Exercise 6: Create an EJB-Based Web Service Client

Task 1 – Create and Run a Client Application

Build a new web service client to test your new EJB web service.

Task 1 - Define an EJB Client Class

1. Create a new Java Application project named `EJBSecuredClient`.

Add support for web service clients to your project. The WSDL for the new web service is located at:
`http://localhost:8080/EJBWebServices/UserManagerSvc?wsdl`

2. Create the `UserManagerEJBClient` class in the `clients` package.
3. Use NetBeans to generate the code to connect your client to the web service.
4. Save the `UserManagerEJBClient.java` file.

Task 2 - Modify the Class to Support Authentication

1. Edit the `UserManagerEJBClient.java` file.
2. Add the following imports.

```
import java.util.Map;
import javax.xml.ws.BindingProvider;
```

3. Add the following code to provide authentication data to the web service.

```
Map<String,Object> reqCtx =
          ((BindingProvider)port).getRequestContext();
reqCtx.put(BindingProvider.USERNAME_PROPERTY, "tracy");
reqCtx.put(BindingProvider.PASSWORD_PROPERTY,"password");
```

4. Save the file.

Task 3 - Test the Client and Web Service

1. Modify the name and email values in your code so test data is added to the database.
2. Build your project. Correct any errors.
3. Run the `UserManagerEJBClient` class and add one user entry.
4. Try using `kelly` to authenticate. Is the user still added?
5. Try using `bubba` or `hacker` to authenticate. What happens?

Task 4 - Shut Down Web Services

Undeploy the web service provider application you deployed in this exercise, and shut down the application server.

Exercise Summary

Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercise.

- Experiences
- Interpretations
- Conclusions
- Applications

Lab 5

Implementing More Complex Services Using JAX-WS

Objectives

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

- Apply JAXB to pass complex objects to and from a web service.
- Map Java exceptions from a web service provider to SOAP faults.
- Inject attributes into JAX-WS web service endpoints.

Exercise 1: Understanding JAX-WS

1. Which of the following is a valid endpoint signature for the WSDL fragment in Code 5-1 below? (Choose one.)

   a. `public void sayHello() throws UDException`

   b. `public String sayHello(String name) throws UDException`

   c. `public String sayHello() throws UDException`

   d. `public void sayHello(String name) throws UDException`

```
<message name =" sayHello">
    <part name =" parameters" element =" tns:sayHello "/>
</message >
<message name =" sayHelloResponse ">
    <part name =" parameters" element =" tns:
    sayHelloResponse "/>
</message >
<message name =" UDException">
    <part name =" fault" element =" tns: UDException "/>
</message >
<portType name =" UserDefinedExceptionWS ">
    <operation name =" sayHello">
        <input message =" tns:sayHello "/>
        <output message =" tns: sayHelloResponse "/>
        <fault name =" UDException" message =" tns:
        UDException "/>
    </operation >
</portType >
```

Code 5-1 Sample Code

2. When do service exceptions occur? (Choose one.)

   a. When a web service call does not result in a fault.

   b. When a web service call results in the service returning a fault.

   c. Whenever a web service call occurs.

   d. Whenever a web service call does not occur.

3. Which parameter(s) is/are required to configure the constructor for `javax.xml.ws.soap.SOAPFaultException`?

   a. fault

   b. fault, actor

   c. fault, actor, code

   d. fault, actor, code, node

4. Which of the following technologies uses `SOAPFaultException` to wrap and manage a SOAP-specific representation of faults?

   a. DOM

   b. SAX

   c. SAAJ

   d. XML

Exercise 2: Add Complex Types to ItemManager

The services implemented so far, ItemManager and UserManager, are limited to simple types. In this exercise, add complex types to the ItemManager web service.

Continue to work on the EJBWebServices project (where you have written EJB-based web services). Extending the EJB-based web services is more straightforward, since your web services can delegate transaction management and security access control to the EJB infrastructure.

Task 1 - Update the **ItemManager** Service API

Update the ItemManager web services API to include complex types as input or output parameters:

1. Add a updateItem operation to ItemManager which accepts an Item and a new description as parameters, to update the persistent version of that item.
   a. Hint: Use dao.find(long var) and item.getId() to get a link to the item in the database.
   b. Examine Model.dao.ejbs to determine how to update the object.
2. Build the web service. Correct any errors.
3. Deploy the updated web service.

Task 2 – Create an **ItemManager** Web Service Client

In this task, create and test a simple authenticating web service for the secured ItemManager web service.

1. Create a new Java Application project named ItemManagerEJBClient.

Add support for web service clients to your project. The WSDL for the new web service is located at:
http://localhost:8080/EJBWebServices/ItemManagerWS?wsdl

2. Create the ItemManagerEJBClient class in the clients package.
3. Save the ItemManagerEJBClient.java file.
4. Add a main method to ItemManagerEJBClient.
5. Add the following imports.

```
import java.util.Map;
import javax.xml.ws.BindingProvider;
```

6. Add the following methods and signatures to the ItemManagerEJBClient class.

```
public static void addItem(){}
public static labs.Item findItem(long itemId){}
public static labs.Item updateItem(labs.Item targetItem
String newDescription){}
```

7. Use NetBeans to generate the code to connect your client to the web service for these methods.

8. Add auththentication support for each method of the `ItemManagerEJBClient` class.

   a. Continue to edit the `ItemManagerEJBClient.java` file.

   b. Add the following code to provide authentication data to the web service.

```
Map<String,Object> reqCtx =
          ((BindingProvider)port).getRequestContext();
reqCtx.put(BindingProvider.USERNAME_PROPERTY, "tracy");
reqCtx.put(BindingProvider.PASSWORD_PROPERTY,"password");
```

9. Test the new `ItemManagerEJBClient` client.

   a. Add a new item to the database by calling the `addItem` method from `main`. Write down the number returned from the add method.

   b. Comment out the previous line of code.

   c. Add code to find the new item just added and change its description.

Exercise 3: Add Complex Types to UserManager

In this exercise, add complex types to the contract-first UserManager web service.

Continue to work on the EJBWebServices project.

Task 1 - Update the UserManager XML Files

Complete the WSDL description of the UserManager web services to include operations that require complex types as output parameters:

- Add a findUser operation to your UserManager, which accepts an id as a parameter, and returns the User with that id.
- Add an updateUser operation to your UserManager, which accepts id, username, and email as parameters (long, String, String). Update the persistent version of that user, and return that User object.

1. Update the XML schema definitions for the web service.
   a. Navigate to the location of the UserManager.xsd file. For the EJBWebServices project, the location under your project is: *Configuration Files -> xml-resources -> web-services -> UserManagerImpl -> wsdl ->* UserManager.xsd.

      After the initial definition of your XML web service, NetBeans move the files to this location. The files must be edited in this location for changes to be made. The WSDL and XSD files in labs project can be removed because they are no longer used.

   b. Add the following XML code to define a User object.

```
<xsd:complexType name="user">
  <xsd:complexContent>
  <xsd:extension base="tns:domainEntity">
    <xsd:sequence>
      <xsd:element name="username" type="xsd:string"  />
      <xsd:element name="email" type="xsd:string"  />
    </xsd:sequence>
  </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>


<xsd:complexType name="domainEntity">
  <xsd:sequence />
```

```
  <xsd:attribute name="id" type="xsd:long" use="required" />
  <xsd:attribute name="version" type="xsd:int" />
</xsd:complexType>
```

The User object can be referred to as tns:user.

c. Using the addUser and addUserReponse as guides, create findUser, findUserResponse, updateUser, and updateUserResponse definitions.

d. Save the updated file.

2. Update the UserManager.wsdl file. It is in the same location as UserManager.xsd.

   a. Using addUserRequest and addUserResp as guides create similar message definitions for the findUser and updateUser operations.

   b. Using addUser operation tag as a guide, create operation tags for findUser and updateUser.

   c. Save the file.

3. Update the UserManagerSvc.wsdl file.

   a. Add new findUser and updateUser operation tags using the addUser operation tag as a guide.

   b. Save the file.

4. Clean and build the project.

Task 2 - Update the UserManagerImpl Class

Add the findUser and updateUser methods to the UserManagerImpl class.

1. Edit the the UserManagerImpl class.
2. Add methods with the following signatures:

```
public auction.User findUser(long id) { }
public auction.User updateUser(long id, String username,
String email){ }
```

Note the return type refers to the User defined by the WSDL files. The auction classes are created when the project builds.

3. The findUser and updateUser methods use Model.User objects to persist data. Modify the methods to copy the contents of the Model.User object returned from a dao operation to auction.User objects. Then return an auction.User object.

4. Save the file.
5. Build the project. Correct any errors.
6. Deploy the project.
7. Confirm that the Web Service has been deployed and running by loading the following URL in the browser:
   `http://localhost:8080/EJBWebServices/UserManagerSvc`

Task 3 - Create a Test Web Service Client

Create a web service client to test the new web service. Name the project `UserManagerEJBClientMod05`. Create the project using one of two approaches.

1. Create a new project using the techniques used in previous exercises.
2. Copy the `SecureEJBClient` project to `UserManagerEJBClientMod05`.
   a. Refresh the `UserManager` web service definitions by refreshing the WSDL file.
      1. Open the `UserManagerEJBClientMod05` project.
      2. Open the *Web Service References* folder.
      3. Right click the `UserManagerSvc` link.
      4. Select *Refresh...*
      5. Make the the correct URL is listed. Check the box that indicates the WSDL file will be replaced.
      6. Click *Yes*.
      7. Rebuild the project.
3. Add new test methods for `findUser` and `updateUser`.
4. Run the updated client application, to test that your new web services are operating correctly.

Task 4 - Shut Down Web Services

Undeploy the web service provider application deployed in this exercise. Shut down the application server.

Exercise 4: Building More Complex Web Services Using JAX-WS

In this exercise, create higher-level services for the auction application, using the code-first approach.

Implement an `AuctionManager` web service. Continue to work in the `EJBWebServices` project.

Task 1 – Design and Implement an **`AuctionManager`** Web Service

An `Auction` application must support, at a minimum, the ability for users to create auctions, look up auctions, and place bids on existing auctions. To simplify the test client, the `AuctionManager` web service returns complex types when a single object is returned. With multiple objects, a String is returned.

1. Define an `AuctionManager` class in the `labs` package in your web service project.
2. Include the operations defined by the following method signatures.

```
public Auction createAuction( long userId, long itemId, int
nDays, double startPrice ) { }
public Auction findAuction( long auctionId ) { }
public Bid placeBid( long userId, long auctionId, double
bidAmount ) { }
public String listAuctions() { }
public String listBids(long auctionId) { }
public Bid getHighBid( long auctionId ) { }
```


3. Set the web service name using the `@WebService` annotation

```
@WebService(serviceName="AuctionManagerService")
```

4. Review the following implementation notes.
   - The `createAuction()`, `findAuction()`, or `listAuctions()` methods use `AuctionDAO` to persist data.
   - The `placeBid()` and `getHighBid()` methods use `AuctionDAO` and `UserDAO` objects and methods to perform their tasks.
   - Remeber to annotate any DAO object with `@EJB`.
5. Deploy the updated web service project.
6. Test the WebService is running at:

```
http://localhost:8080/EJBWebServices/AuctionManagerService
```

Optional

Add the ability for the new web service to authenticate users:

- Declare that all may call any operations on `AuctionManager`, using annotations on the Java class.
- Remember that the `@PermitAll` annotation captures this constraint.
- Require that the container obtain identity information for all callers of the `AuctionManager` service.

This involves modifying the `web.xml` configuration file for the web application. Add he URL used to contact the `AuctionManager` web service to the list of protected resources that require container level authentication. Remember to limit authenticated access to this URL and POST requests. Set the request to obtain the web service's description directly from the web service to always allowed (GET request).

Task 2 – Create an **`AuctionManager`** Web Service Client

Create a client to test the `AuctionManagerSevice` web service.

1. Create a new web service client project named `AuctionManagerClient`. Name the client class `AuctionManagerClient`.
2. Perform the following operations as part of your testing.
   a. Create an auction.
   b. Find an auction.
   c. List all auctions.
   d. Place bids on an auction.
   e. Get a list of all bids on an auction.
   f. Get the high bid for an auction.

3. Run your new client application, to test the `AuctionManagerService` web service.

Note – Leave the server application running. The next lab requires the `EJBWebServices` application to be running and loaded with data from this lab

Exercise Summary

Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercise.

- Experiences
- Interpretations
- Conclusions
- Applications

Lab 6

JAX-WS Web Service Clients

Objectives

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

- Create web service clients using JAX-WS.
- Create web service clients using JAX-WS that support asynchronous pull-based interactions.

Exercise 1: Understanding How to Create Web Service Clients

1. Which is the proper annotation to inject a Service instance into a JavaEE web service client? (Choose one).
   a. `@WebService`
   b. `@WebServiceRef`
   c. `@Resource`
   d. `@EJB`
2. Which is the proper way to enable schema validation on a JAX-WS client? (Choose one.)
   a. Add a binding declaration to the WSDL description of the service.
   b. Add the `@SchemaValidation` annotation to the web service client implementation class.
   c. Create an instance of the proper `WebServiceFeature` subclass, then pass it to the `getPort()` call on a Service.
   d. There is nothing to do: schema validation is enabled by default on JAX-WS web service clients.
3. Which is the proper way to enable support for asynchronous interactions in a JAX-WS web service port proxy? (Choose one.)
   a. Add the `@Asynchronous` annotation to the method to invoke asynchronously on the web service implementation class.
   b. Add the `@Asynchronous` annotation to the field that will hold the reference to the JAX-WS port proxy on the client side.
   c. Add a binding customization to the WSDL description of the web service, on the client side.
   d. One cannot invoke asynchronous operations at the level of the JAXWS proxy type; the only way to invoke an operation asynchronously in JAX-WS involves bypassing the proxy and using the dynamic JAXWS Dispatch API.

Exercise 2: Building an Asynchronous Web Service Client

Task 1 – Creating an Asynchronous Web Service Client

All the client applications so far assume nothing else is going on in the client at the time a call is made to the web service. Therefore, it is reasonable for the client to simply issue the call and wait for the answer to come back.

A more sophisticated `Auction` client could present its user with status information on multiple auctions concurrently. A client could also update the status displayed for the auction that the user is currently interested in, even in the middle of performing other operations.

In this exercise, write a client for the `AuctionManager` web service that issues a request to get the highest bid on an auction without blocking while the request to place that bid is processed on the server side.

1. Continue to work with the `EJBWebServices` project.
2. Create a new Java Application project and name it `AsyncClients`.
3. Add web service client code to the project just like previous labs. Use the `AuctionManagerService` url to define the service.

   ```
   http://localhost:8080/EJBWebServices/AuctionManagerService?wsdl
   ```

4. Add support for asynchronous operations to your project.
   a. Click the node in the *Project* tab.
   b. Click *Web Service References* folder and open it.
   c. Right click the `AuctionManagerService` node.
   d. Select *Edit Web Service Attribute*s.
   e. Click the *WSDL Customization* tab.
   f. Open the *Global Customization* node.
   g. Check *Enable Asynchronous Client*.
   h. Click OK.

   Asynchronous support is now enabled in this project. The project will rebuild itself to enable this.

NetBeans Tool Reference Guide: Java EE Development > Web Services > JAX-WS Web Services > Editing Web Service Attributes

5. Create a new client class for the `AuctionManager` web service named `AsyncClient.java`.
6. Use the NetBeans wizard to insert new code into your `main` method.

   The dialog offers a list of all the operations available on the `AuctionManager` web service including the asynchronous variants of each of the operations.
7. Select the polling asynchronous option to get the high bid.
8. Have the client print a message between the operation request and the display of the result.
9. Run the client application, to test this asynchronous interaction.

Task 2 – Shut Down Web Service

Undeploy the web service provider application deployed in this exercise, and shut down the application server.

Exercise Summary

Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercise.

- Experiences
- Interpretations
- Conclusions
- Applications

Exercise Summary

Lab 7

Introduction to RESTful Web Services

Objectives

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

- Describe what RESTful Web Services are.
- List the five principles behind RESTful Web Services.
- Describe the advantages and disadvantages of a RESTful approach.

Exercise 1: Understanding RESTful Web Services

1. Indicate whether each of the following assertions is true or false:
   a. RESTful web services are procedural, rpc-oriented services.
   b. RESTful web services prefer to minimize the number of interactions between client and server, and so prefer to deliver complex data representations.
   c. RESTful web services commit to XML-based representations for communications.
   d. RESTful web services prefer stateless interactions.
   e. RESTful web service requests represent the operation to be performed somewhere in the URI used to invoke that operation on the server side.
2. Choose the statement that best describes the RESTful approach to choosing the right HTTP method to use for any request:
   a. GET is always the preferred method to use.
   b. POST is to be preferred over PUT, as it is more flexible.
   c. PUT is to be preferred over POST, as it is more precise.
   d. POST and PUT have different semantics, so the choice must be based the semantics of the operation.
3. Fill in the HTTP method that corresponds to each of the semantics quoted.,

Purpose	Method
Read, possibly cached	
Update or create without a known ID	
Update or create with a known ID	
Remove	

Exercise Summary

Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercise.

- Experiences
- Interpretations
- Conclusions
- Applications

Exercise Summary

Lab 8

RESTful Web Services: JAX-RS

Objectives

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

- Implement RESTful web services using JAX-RS
- Deploy REST web services using Jersey, an implementation of JAX-RS

Exercise 1: Understanding How to Define Web Services Using JAX-RS

1. Indicate whether each of the following statements is true or false:
   a. Every JAX-RS root resource class must be annotated with an `@Path` annotation.
   b. If a public method of a resource class does not include an explicit HTTP method annotation, JAX-RS will act as though the code had specified `@GET`.
   c. Just like web containers do with servlets, the JAX-RS runtime will create an instance of a root resource class the first time the runtime needs that class to handle a client request, then reuse that same instance as other clients require access to the same root resource class.
   d. A class that inherits from Application is mandatory for every server side application based on JAX-RS, so as to configure the set of resources offered by that application.
   e. JAX-RS guarantees thread-safety for root resource classes annotated with the `@Singleton` annotation.
2. Two resource methods on the same resource class must differ in one of the following: (Choose all that apply)
   a. `@Path` URI template.
   b. HTTP Method.
   c. Representations produced.
   d. Representations consumed.
   e. Visibility.
   f. Arguments (in number or types).
3. Choose the name of the type of object that you can inject into a JAX-RS resource class to help it build URIs relative to the current one, to be used as part of responses. (Choose one).
   a. PathInfo
   b. UriInfo
   c. Context
   d. Response

Exercise 2: Create, Deploy, and Test the `ItemManager` Web Service

In this exercise create, deploy, and test a simple RESTful web service for the `Auction` application. This application is a counterpart to the first simple JAX-WS service you implemented before. Once the web service is ready, deploy it to a stand-alone Java VM. Since this is a RESTful web service, test it using a browser.

RESTful web services rely on specific HTTP methods to perform operations. The Live HTTP Headers plugin for Firefox (`live_http_headers-0.16-fx+sm.xpi`) displays all the HTTP headers involved in an HTTP conversation. The plugin is provided in the `resources\lab08` directory. To install, just navigate to the file and select it using the Firefox *File -> Open* menu. The plugin can also be downloaded from:
`https://addons.mozilla.org/en-US/firefox/addon/3829/`

Task 1 – Define URIs Required by RESTful Service

Imagine an Item manager web service which allow its clients to create, find, and manage a set of persistent `Item` descriptions. A similar system to the JAX-WS `ItemManager` service created earlier.

1. To design a RESTful architecture, identify the URIs used to refer to the entities exposed through the RESTful service. Remember, URIs identify the entity of interest, not the operations performed with that entity.
2. Next, choose the operations supported by this RESTful service. In a RESTful architecture, operations are mapped to standard HTTP methods invoked on the entities exposed through this service.

Table 8-1 Requests Supported by RESTful Item Manager

Request	HTTP Method	URI
Add New Item		
Find Item		
Remove Item		

Table 8-1 allows you to capture the RESTful request that represent the operations that your web service supports.

Consider whether the requests that accept parameters describe them as components in the URI path, or query or form parameters to the request.

Task 2 – Define POJO Class

Implement a JAX-RS `ItemManagerRS` service. The service creates new `Item` instances and finds existing `Item` instances that match a new item. The service removes instances that are no longer used. Limit the API to accept primitive types and Strings. The methods should return Strings with one exception.

1. Create a new Java Application project in NetBeans named `RESTfulServices`. Use this project to create a stand-alone JAX-RS (RESTful) web service.
2. Add the JAX-RS 1.1 and Jersey 1.1 (JAX-RS RI) libraries to the project to have access to the JAX-RS APIs and Jersey implementation.
3. Add the dependencies required to support server-side operations for the `Auction` application.
   a. `AuctionApp`
   b. EclipseLink
   c. DerbyJDBC
4. Create a new class `ItemManagerRS`, in package `labs`.
5. Add the following methods to the class.

```
public String addItem(String description) { }
public String formAdd(String description) { }
public String removeItem(long id) { }
public Item findItem(long id) { }
```

   Notice for the `findItem` method an `Item` is returned. How will the web service handle this? You can see the answer when you test the lab.

6. Add a DAO object to persist and retreive data.
7. Implement the business logic for each method in `ItemManagerRS`. Return any String messages in HTML tags (e.g., p, h3, h4).

Note – Consider copying the contents of the `ItemManager` class from Lab 2 Exercise 2 (use the Solution if you wish). Remove any JAX-WS annotations present in the code. Do not copy the `ItemManagerRunner` class from the earlier project.

8. Compile the class. Correct any errors.

Task 3 – Incorporate JAX-RS Annotations

Incorporate JAX-RS annotations to turn `ItemManagerRS` into a RESTful web service.

1. Add the following imports to support JAX-RS annotation.

```
import javax.ws.rs.DELETE;
import javax.ws.rs.FormParam;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
```

2. Add an `@Path("/items")` annotation at class level. This indicates that `/items` is the root resource for this web service.
3. Add the following `@Path` annotations and method annotations for each class.
   a. `addItem` - `POST` method with a path of `/{description}`
   b. `formAdd` - `POST` method with a path of `/add`
   c. `removeItem` - `DELETE` method with a path of `/{id}`
   d. `findItem` - `GET` method with a path of `/{id}`
4. Add `@PathParam` annotations to each method signature to extract parameters that are encoded as part of the URI.
   a. `addItem` - `description`
   b. `removeItem` - `id`
   c. `findItem` - `id`
5. Add an `@FormParam` annotation to the `formAdd` method signature. Add `description` as the parameter value.
6. Compile the class. Correct any errors.

Task 4 – Deploy Web Service to Stand-alone JVM

Deploy your `ItemManager` web service in a stand-alone JVM:

1. Add a new class named `Runner` in the `labs` package. Implement the class to deploy your web service.

Jersey provides a class called `HttpServerFactory` (which can be found in package `com.sun.jersey.api.container.httpserver`). The class sets up a stand-alone JAX-RS service in a Java web server.

2. Add the following imports to your class.

```
import
com.sun.jersey.api.container.httpserver.HttpServerFactory;
import com.sun.net.httpserver.HttpServer;
import java.io.IOException;
```

3. Add a `main` method.
4. Add the following code to the `main` method to launch your web service.

```
String url = "http://localhost:8081/jaxrs";
HttpServer server = HttpServerFactory.create( url );
server.start();
```

5. Compile the new class. Correct any errors.
6. Ensure that the database server is running.
7. Run the `Runner` class.

Task 5 – Test Web Service Using Browser

RESTful web services represent web service interactions in terms of simple HTTP requests and responses. A web browser is enough to generate the requests that the RESTful web service processes.

1. Make sure the JavaDB database is started.
2. Add items to the database.
   a. To do this use one of the forms provided in the `resources\lab08` directory.
   b. For example, the `AddGuitar.html` file adds items using the `addItem` method defined in the `ItemManagerRS` class.

```
<html >
<body >
<form action="http://localhost:8081/jaxrs/items/Guitar"
method="POST">
<input type="submit"/>
</form >
</body >
</html >
```

c. The `AddItem.html` file adds items using an HTML form and the `formAdd` method.

```
<html >
<body >
<form action="http://localhost:8081/jaxrs/items/add"
method="POST">
<input type="text" name="description"/><br/>
<input type="submit"/>
</form >
</body >
</html >
```

d. Check the database, several items should be added.

3. To retreive the items added to the database using the following URL.

   `http://localhost:8081/jaxrs/items/1`

4. Replace 1 with the ID of any row to retreive. What format are the results delivered in?

Note – Sample HTML forms for testing this lab and the remaining labs is provided in the `resources\lab08` directory.

Task 6 – Shut Down Web Service

Shut down the `ItemManagerRS` application.

Exercise 3: Create, Deploy, and Test the `UserManager` Web Service

In an earlier lab, you built a `UserManager` web service using the contract-first approach. This approach is not compatible with a RESTful architecture.

In this exercise, implement the RESTful counterpart to the `UserManager` service. The activities that the web service would need to support, as far as user management is concerned, include:

- Add new users to the application.
- Update information about an existing user, such as their full name or their email address.
- Find users added to the application.

Task 1 – Define URIs Required by RESTful Service

The `UserManagerRS` web service allows its clients to add, find and update the set of persistent `User` instances representing users known by the system:

1. In a RESTful architecture, identify the URIs used to refer to the entities to be exposed through the RESTful service. Remember that URIs identify the entity of interest, not the operations to be performed with that entity.
2. Choose which operations will be supported by this RESTful service. In a RESTful architecture, those operations are mapped to the standard HTTP methods invoked on of the entities exposed through this service.

Use Table 8.2 to capture the RESTful request that represents the operations the web service supports.

Table 8-2 Requests Supported by RESTful User Manager

Request	HTTP Method	URI
Add New User		
Update User		
Find User		

Consider whether the requests that accept parameters should describe them as components in the URI path, or query or form parameters to the request.

Task 2 – Define POJO Class

Update the `RESTfulServices` project to support the `UserManager` web service.

1. Create a new class named `UserManagerRS` in `labs` package.
2. Add the following methods to the class.

```
public String addUser( String name, String email ) { }
public User findUser( long id) { }
public String updateUser( long id, String name, String
email) { }
```

3. Implement the business logic associated with each of the functions in the interface for `UserManagerRS`.
4. Compile the class. Correct any errors.

Note – You can use the code written for `UserManagerImpl` in an earlier lab as a starting point for this class.

Task 3 – Incorporate JAX-RS Annotations

Incorporate JAX-RS annotations to turn `UserManagerRS` into a RESTful web service.

1. Add an `@Path("/users")` annotation at class level. This indicates that `/users` is the root resource for this web service.
2. Add the following `@Path` annotations and method annotations for each class.
   a. `addUser` - `POST` method with a path of `/add`
   b. `findUser` - `GET` method with a path of `/{id}`
   c. `updateUser` - `POST` method with a path of `/update`. The `POST` annotation is only used because HTML forms only support `POST` and `GET`. An update operation in REST should use the `PUT` method.
3. Add `@FormParam` annotations to the following methods.
   a. `addUser` - `name, email`
   b. `updateUser` - `id, name, user`

4. Add an `@PathParam` annotation to the `findUser` method signature. Add `id` as the parameter value.
5. Compile the class. Correct any errors.

Task 4 – Deploy Web Service to Stand-alone JVM

Deploy your `UserManagerRS` web service to a stand-alone JVM. No new code is needed for this. The `Runner` class deploys each web service at the URL indicated by the annotations.

Task 5 – Test Web Service Using Browser

Test the new web service using a web browser as before.

1. Make sure the JavaDB database is started.
2. Add users to the database.
   a. To do this use one of the forms provided in the `resources\lab08` directory.
   b. The `AddUser.html` file adds items using an HTML form and the `addUser` method.

```
<html>
  <head><title>Add User</title></head>
<body>
<form action="http://localhost:8081/jaxrs/users/add"
method="POST">
<label>Name: </label>
<input type="text" name="name"/><br/>
<label>EMail: </label>
<input type="text" name="email"/><br/>
<input type="submit"/>
</form>
</body>
</html>
```

   c. Check the database to make sure users are added.
3. To retreive the items added to the database using the following URL.

   `http://localhost:8081/jaxrs/users/1`
4. Update a user.

a. The `UpdateUser.html` file updates an existing user using an HTML form and the `updateUser` method.

```
<html>
<head><title>Update User</title></head>
<body>
<form action="http://localhost:8081/jaxrs/users/update"
method="POST">
<label>ID: </label>
<input type="text" name="id"/><br/>
<label>New Name: </label>
<input type="text" name="name"/><br/>
<label>New EMail: </label>
<input type="text" name="email"/><br/>
<input type="submit"/>
</form>
</body>
</html>
```

Exercise Summary

Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercise.

- Experiences
- Interpretations
- Conclusions
- Applications

Lab 9

JAX-RS Web Service Clients

Objectives

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

- Create web service clients using JAX-RS.

Exercise 1: Understanding RESTful Clients

1. Indicate whether each of the following statements is true or false:
   a. Clients of a RESTful service that are implemented using the class java.net.URL in Java are limited to HTTP GET requests, because the URL class has that limitation.
   b. `java.net.URL` cannot pass an entity body as part of any request.
   c. The machinery associated with `java.net.URL` can automatically encode and decode query parameters and returns encoded in Base-64.
   d. The machinery provided by the Jersey Client API can automatically encode and decode query parameters and returns encoded in Base-64.
2. Choose the type used to represent RESTful resources in the Jersey Client API: (Choose one).
   a. Resource
   b. Client
   c. WebResource
   d. URL
3. To obtain metadata about the most recent interaction with the web service, the Jersey Client API requires that:
   a. The client specify that the expected type for the incoming payload be `ClientResponse`.
   b. The client call a `getMetadata()` API in WebResource.
   c. The client call the `WebResource` HTTP method within a `try/catch`, because the metadata will be part of the exception that the Jersey runtime will throw to the application.
   d. The client gain access to the underlying `HTTPUrlConnection` instance because HTTP metadata is not exposed through the Jersey Client API.

Exercise 2: Building Web Service Clients

In this lab, build web service clients for the `ItemManagerRS` and `UserManagerRS` web services using the Jersey client APIs.

Task 1 – Create New Project

Create a new project for the clients.

1. Create a new Java Application project called `RSClients`.
2. Add the following libraries and projects to the new project.
   a. The `AuctionApp` project.
   b. JAX-RS 1.1
   c. Jersey 1.1 (JAX-RS RI)

Task 2 – Build a Web Service Client for **`ItemManagerRS`**

Build a client for the `ItemManagerRS` web service.

1. Create a `clients` Java package.
2. Create a Java class named `ItemManagerClient.`
3. Create a `main` method.
4. Add the following imports.

```
import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.ClientResponse;
import com.sun.jersey.api.client.WebResource;
import com.sun.jersey.core.util.MultivaluedMapImpl;
import javax.ws.rs.core.MultivaluedMap;
```

5. In the `main` method, add the following Jersey client API code.
   a. Obtain an instance of `Client`.

```
Client client = Client.create();
```

b. Obtain instances of `WebResource` that represent the resources to contact.

```
String url = "http://localhost:8081/jaxrs/items/add";
WebResource resource = client.resource( url );
```

c. Add parameters for adding an item.

```
// Add parameters
MultivaluedMap<String, String> params = new
MultivaluedMapImpl();
params.add( "description", "Guitar" );
```

d. Invoke the `WebResource` and print the results. Notice the HTTP method type is determined by a Java method.

```
String result =
        resource
        .type( "application/x-www-form-urlencoded" )
        .post( String.class, params );
      System.out.println( result );
```

6. Build the client. Correct any errors.
7. Run the client application, to test all the add operation in `ItemManagerRS`.
8. User your browser to test the data added to the web service. For example:

```
http://localhost:8081/jaxrs/items/1
```

9. Stop the `RESTfulServices` project when finished.

Task 3 – Build a Web Service Client for **`UserManager`**

Build a client for the `UserManagerRS` web service.

1. Create a Java class named `UserManagerClient`.
2. Add a `main` method.
3. Using the previous task as an example, provide the code necessary to implement a `addUser`, `findUser` and `updateUser` methods.
   a. To use the GET HTTP method for the findUser method use code like this:

```
ClientResponse response =
resource.get(ClientResponse.class);
System.out.println( response.getEntity(String.class) );
```

4. Update the `RESTfulServices` project so the UserManagerRS uses a PUT annotation instead of a POST annotation. Rebuild the project and deploy the web service.
5. Run the client class (use Shift-F6 to run an individual file) to test all the operations in `UserManagerRS`.

Note – Be sure to record the ID returned from any add operation. JavaDB may not always assign 1 to the first row added. It could assign 51 or some other number. Passing the find or update operations the wrong ID will result in exceptions.

Exercise Summary

Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercise.

- Experiences
- Interpretations
- Conclusions
- Applications

Lab 10

JAX-RS and JavaEE

Objectives

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

- Deploy RESTful web services to a web container.
- Secure RESTful web services in a web container, delegating authentication to the container.

Exercise 1: Understanding RESTful Resources in a Container

1. Indicate whether each of the following statements is true or false:
   a. Deploying RESTful JAX-RS resources in a web application requires changes to the `web.xml` configuration file.
   b. Deploying RESTful JAX-RS resources in a web application requires the presence of a subclass of `Application`.
   c. Security constraints that apply to a JAX-RS resource can only be captured in the `web.xml` configuration for the web application that the resource is deployed in.
   d. Declarative annotation of security constraints as annotations on the resource class automatically configures the web container's login configuration for this application.
   e. Declarative authorization constraints can be specified in terms of specific users of the application.
   f. The JAX-RS and JAX-WS specifications are not compatible, so the only way to have web services that can be reached via either REST or SOAP is to have two separate implementation classes, one implemented in JAX-RS, and the other in JAX-WS.
   g. A JAX-RS resource that is implemented as a Singleton EJB must be made thread-safe programmatically at application level.
2. Choose the types that can be injected into a JAX-RS resource class to obtain information about the caller of a service: (Choose as many as appropriate.)
   a. `SecurityContext`
   b. `ServletContext`
   c. `UriInfo`
   d. `ServletRequest`

Exercise 2: Deploy RESTful Web Services to a Web Container

In Lab 8, you built two simple web services, `ItemManagerRS` and `UserManagerRS`. Both are simple RESTful POJO web services and are deployed stand-alone in a JVM. In this exercise, deploy the web services into a web application and leverage the functionality provided by the web container.

Task 1 – Copy Existing Web Services to Web Application

RESTful POJO web services can be deployed either stand-alone, using the HTTP server built into Java SE 6, or to a web container. The implementation of the POJO web service itself does not have to change because of the environment it is deployed in. Any differences are handled by the JAX-RS (Jersey) runtime.

1. Create a new Web Application project named `RSInContainer`.
2. Add the following libraries to the project.
   a. The `AuctionApp` project.
   b. EclipseLink (JPA 2.0)
   c. Derby JDBC libraries.
   d. JAX-RS 1.1.
   e. Jersey 1.1 (JAX-RS RI).
3. Configure a database connection pool as described in Lab 4, Exercise 2, Task 2. Name the pool `auctionDbPool` and use the NetBeans connection to the `Auction` database to set up the pool.
4. Configure a JDBC resource as described in Lab 4, Exercise 2, Task 2. Connect the database through the `auctionDbPool`. Name the JDBC resource `jdbc/auction`.
5. Copy the code for the two services `ItemManagerRS` and `UserManagerRS` from the RESTfulServices project you created in Lab 8 into this new project.

   The simplest way to copy the two services is:

   a. Select the labs package from the original `RESTfulServices` project.
   b. Right-click that `labs` node, and select *Copy*.
   c. Select the *Source Packages* node within the `RSInContainer` project.
   d. Right-click that *Source Packages* node, and select *Paste*.

When the two JAX-RS implementation classes are copied into the `RSInContainer` project, NetBeans pops up a dialog, Figure 10-1.

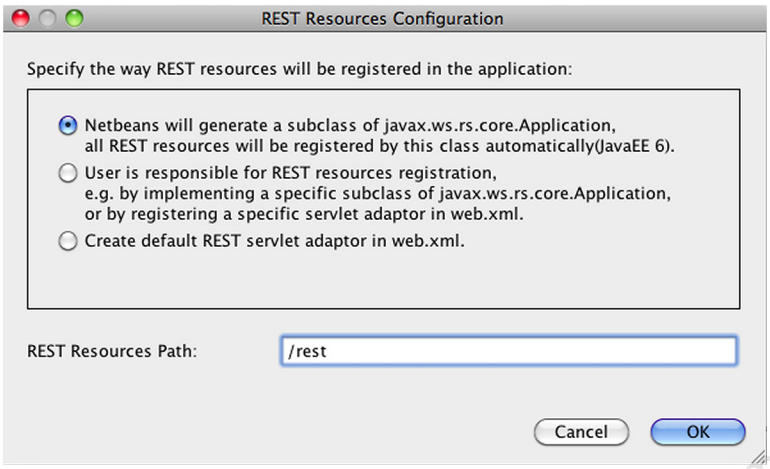

Figure 10-1 Automatic JAX-RS Configuration Dialog on Import

e. Select the first option. This lets NetBeans configure these resources automatically.

f. Change the resource path to `/rest` to match the instructions in the next step.

g. Click *OK*.

A new node named `RESTful Web Services` appears in the `RSInContainer` project. It contains descriptions of the RESTful services NetBeans knows exist within the project.

NetBeans also creates a node called *Generated Sources (rest)* within the `RSInContainer` project. Examine the `ApplicationConfig` class within this node. The class contains an `@ApplicationPath` annotation to specify the prefix that distinguishes URIs that refer to RESTful services within this application. In this case, the path is `/rest`.

Note – Delete the `Runner` class that was also copied above. It is no longer needed in a web container.

6. Copy the HTML forms from the previous project into the web directory. You can use the forms to add data to your web service.

   By default, the URLs take the following form:

   `http://localhost:8080/RSInContainer/pathInAnnot`

As a result of the selections made when the web service was copied, the two web services have the following format.

```
http://localhost:8080/RSInContainer/rest/pathInAnnot
```

7. Modify your HTML forms to reflect the new path to the web services.

Task 2 – Configuring Web Services Within Web Application

No explicit configuration changes to the `web.xml` file are needed. JAX-RS configures URLs to access the web services within the application by default, based on their `@Path` annotations.

Task 3 – Deploy and Test Services in Web Application

Deploy the `RSInContainer` web application to a web container. The RESTful web services contained within it are automatically deployed.

To test the services, repeat the tests from the original stand-alone services in Labs 8 and 9:

- Use a web browser and HTML forms to test each web service. [If you want to use the `UpdateUser.html` form, the HTTP method needs to be set to POST in the web service code.]
- [Optional] Write simple Java client applications that use the Jersey Client API to invoke operations on these services.

Note – Be sure the stand-alone JAXRS service applications used in Lab 8 are shut down.

Exercise 3: Secure POJO Web Services in Web Container

In this exercise, secure access to the `UserManagerRS`, so that only administrators can create new users.

Task 1 – Add Security Annotations

There are two kinds of users of the `UserManagerRS` web service, administrators and everyone else. Only administrators are allowed to add new users to the application.

The way to configure security constraints in JAX-RS web services borrows the same annotations that JAX-WS (and before it, EJBs) use to capture these constraints.

1. There are two roles `user` and `administrator`. At the class level, add the `@DeclareRoles` and `@RolesAllowed` annotation to both web service source files.
2. At the method level, at a `@RolesAllowed({"administrator"})` annotation to the `addUser` method of `UserManagerRS`.

Task 2 – Define Security Constraints

Configure `web.xml` to support authentication for the JAX-RS web services. The configuration required for JAXWS and JAX-RS are the with one exception: JAX-RS web service requests can be triggered by any HTTP method.

1. Add a `login-config` element. Set authentication to `BASIC` which requires clients to provide userid and password information in an HTTP header.

```
<login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>file</realm-name>
</login-config>
```

2. Define the user and administrator security roles for the web service.

```
<security-role>
        <description/>
        <role-name>user</role-name>
</security-role>
<security-role>
        <description/>
```

```
        <role-name>administrator</role-name>
</security-role>
```

3. Add a `security-constraint` element to identify the HTTP methods and roles allowed to access the web service. In addition, specify the path that needs to be constrained.

```
<security-constraint>
        <display-name>AuctionServices</display-name>
        <web-resource-collection>
            <web-resource-name>AuctionServices</web-
resource-name>
            <description/>
            <url-pattern>/rest/*</url-pattern>
            <http-method>GET</http-method>
            <http-method>POST</http-method>
            <http-method>PUT</http-method>
            <http-method>DELETE</http-method>
        </web-resource-collection>
        <auth-constraint>
            <description/>
            <role-name>administrator</role-name>
            <role-name>user</role-name>
        </auth-constraint>
    </security-constraint>
```

4. Configure `sun-web.xml` security constraints. Add role mappings for user `kelly` and administator `tracy`.

```
<security-role-mapping>
    <role-name>administrator</role-name>
    <principal-name>tracy</principal-name>
  </security-role-mapping>
  <security-role-mapping>
    <role-name>user</role-name>
    <principal-name>kelly</principal-name>
  </security-role-mapping>
```

Task 3 – Configure Users in Authentication Realm

You already configured `kelly` (user) and `tracy` (administrator) in the application server. Continue to use the same set of users as before.

Task 4 - Test Authentication in Browser

Test the new web service using a web browser as before.

1. Deploy `RSInContainer` web service project.
2. Add items to the database using `AddItem.html` or `AddGuitar.html`. You should be prompted for username and password the first time you access the web service.
3. Check the database to make sure items are added.
4. To retreive the items added to the database using the following URL.

   ```
   http://localhost:8080/RSInContainer/rest/items/1
   ```

5. Add users to the database using `AddUser.html`. You should be prompted for username and password the first time you access the web service. Only `tracy` should be allowed to add users.
6. Check the database to make sure users are added.
7. To retreive the items added to the database, use the following URL.

   ```
   http://localhost:8080/RSInContainer/rest/users/1
   ```

Task 5 - Build Authenticating Web Service Clients

In lab 9, you created two stand-alone Jersey client API applications to interact with RESTful services, `ItemManagerClient` and `UserManagerClient`. Use those two applications to test the web services deployed as part of the `RSInContainer` web application. In this task, add authentication support to the two clients.

1. Open the `RSClient` project created in Lab 9.
2. Copy the two client classes into two new classes (java files in the same labs package), named `AuthItemManagerClient` and `AuthUserManagerClient`.
3. Modify `AuthItemManagerClient` class to use the new container URI. For example:

   ```
   http://localhost:8080/RSInContainer/rest/items/add
   ```

4. Modify `AuthUserManagerClient` class to use the new container URI. For example:

   ```
   http://localhost:8080/RSInContainer/rest/users/add
   ```

5. Modify the two new applications to provide authentication information when requests are issued to the web services.

a. Add an import for the `ClientFilter`.

```
import com.sun.jersey.api.client.filter.ClientFilter;
```

b. Add the following code to create the `WebResource` with authentication information.

```
String username = "tracy";
String password = "password";

ClientFilter authFilter =
            new HTTPBasicAuthFilter(username,password);
client.addFilter(authFilter);
WebResource resource = client.resource( url );
```

c. Add this code wherever necessary in the client to add authentication features.

Task 5 – Deploy and Test Services in Web Application

1. Make sure the `RSInContainer` web application is deployed.
2. Run your two authenticating client applications, to test that the web services are working correctly, and limiting access to the right users.

Note – Be sure to record the ID returned from any add operation. JavaDB may not always assign 1 to the first row added. It could assign 51 or some other number. Passing the find or update operations the wrong ID will result in exceptions.

Exercise Summary

Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercise.

- Experiences
- Interpretations
- Conclusions
- Applications

Lab 11

Implementing More Complex Services Using JAX-RS

Objectives

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

- Pass complex objects to and from a web service.
- Map Java exceptions from a web service provider to SOAP faults.

Exercise 1: Understanding Features of JAX-RS

1. Indicate whether each of the following statements is true or false:
   a. Jersey only supports XML representations out of the box, through Java's built-in JAXB technology. Other representations have to be added as application-level extensions, via classes that implement the two JAXRS interfaces `MessageBodyReader` and `MessageBodyWriter`.
   b. Any one resource method can only support one representation type.
   c. `Response` objects are used to capture metadata about the return value to be delivered to the caller.
   d. A `resource` method that needs to return metadata to a caller, when an exception is thrown within the `resource` method, must be defined to return a `Response` instance so the method can return the appropriate metadata.
   e. When creating a custom representation for an application-level type, both `MessageBodyReader` and `MessageBodyWriter` implementations for that application-level type must be provided.
   f. Instances of entity *provider* types (classes that implement the two interfaces `MessageBodyReader` and `MessageBodyWriter)` are created and thrown away after each use.
2. Choose the class-level annotation that must be used for any classes that implement `MessageBodyReader`, `MessageBodyWriter`, or `ExceptionMapper`: (Choose one).
   a. `@Context`
   b. `@Path`
   c. `@Provider`
   d. `@Resource`
3. How can you identify sub-resource locator methods? (Choose the one best match).
   a. They are methods in a `resource` class that have no `@Path` annotation.
   b. They are methods in a `resource` class that have no HTTP method annotation.
   c. They are methods in a resource class that are annotated with @Resource.
   d. They are private methods in a `resource` class.

Exercise 2: Improve Web Service Responses

RESTful web services are expected to model the HTTP standards for web interaction. In particular, resource responses carry status codes along with any payload for the client in order to give the client more information (metadata) about the request just processed.

In this exercise, update your JAX-RS `UserManagerRS` web service to provide that additional metadata.

- Requests to add users ought to return a `201 - Created` status code, along with the URI for the client just created.
- Requests to find a user returns a status code of `200 - OK`.
- When a user is not found, return a `404 - Not Found` status code.
- Return a `304 - Not Modified` status code if the update operation did not actually change the user specified.
- Return a `303 - See Other` status code to redirect the client to the updated user entry when the update suceeds.

Task 1 - Add Responses to the Web Service

Update a previous project to add response features to the web service. You may optionally use your existing project instead of the solution upon approval of your instructor. All the web service methods now return `Response` objects.

1. Copy the `RSInContainer` project solution from Lab 10 exercise 2 into a new directory.
2. Open the `UserManagerRS` class file.
3. Modify the `findUser` method to return `404 - Not Found` or the user object and a `200 - OK` response. For example:

```
if (user == null) {
return Response.status(Response.Status.NOT_FOUND).build();
} else {
return Response.ok(user).build();
}
```

4. Modify the `addUser` method to return a `201 - Created` status code and the URI location of the new user. For example, here is the code for the created response.

```
return Response.created(location).build();
```

a. In the previous example, `location` is defined as a URI. For example:

```
URI location = new URI("http://localhost:8080/RSInContainer/rest/users/"
+ user.getId());
```

5. Modify the `updateUser` method to return `404 - Not Found` if the user cannot be found. If the user cannot be updated return `304 - Not Modified`. If the operation is successful, return `303 - See Other` and redirect the client to the new user entry. For example:

```
return Response.seeOther(location).build();
```

a. The location variable is a URI as discussed before.

Task 2 - Test the Updated Web Service

Using your browser and the HTML forms, test the new web service.

Exercise Summary

Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercise.

- Experiences
- Interpretations
- Conclusions
- Applications

Exercise Summary

Lab 12

Trade-Offs Between JAX-WS and JAXRS Web Services

Objectives

On completion of this lab, you should be able to:

- Discuss the trade-offs involved in the choice to implement a web service using either JAX-WS or JAX-RS technology.

Exercise 1: Understanding the difference between JAX-WS and JAX-RS technologies

1. Choose the qualifier that best describes each of the two types of web services in the table below. (Choose one row for each column).

	SOAP Services	REST Services
Activity-oriented		
Method-oriented		
Resource-oriented		
URI-oriented		

Exercise 2: Building More Complex Web Services Using JAX-RS

The goal of the exercises is to create web services that support an `Auction` application. In this exercise, write higher-level RESTful services associated with such an `Auction` application.

Task 1 - Define the Web Service Methods

Define the implementation methods for a RESTful `Auction` web service.

1. Continue to work in the `RSInContainer` project from the previous lab.
2. In the `labs` package, create the `AuctionMangerRS` class.
3. The web service methods need to support the following operations: create an auction, find an auction, place a bid, list all auctions, list all bids on an auction, and get the high bid for an auction.
4. Implement all those operations in your web service. Below are sample method signatures and annotations provided for your convenience.

```
@POST @Path("/create")
public Response createAuction( @FormParam("userId") long userId,
@FormParam("itemId") long itemId, @FormParam("nDays") int nDays,
@FormParam("startPrice") double startPrice ) throws URISyntaxException{ }

@GET @Path("/{id}")
@Produces("application/xml")
public Response findAuction( @PathParam("id") long auctionId ) { }

@POST @Path("/bid/place")
@Produces("application/xml")
public Response placeBid( @FormParam("userId") long userId,
@FormParam("auctionId") long auctionId, @FormParam("bidAmount") double
bidAmount ) { }

@GET @Path("/list")
@Produces("application/xml")
public Response listAuctions() { }

@GET @Path("/{id}/listbids")
@Produces("application/xml")
public Response listBids(@PathParam("id") long auctionId) { }
```

```
@GET @Path("/{id}/highbid")
@Produces("application/xml")
public Response getHighBid( @PathParam("id") long auctionId ) { }
```

5. Some of the operations may return a `List` of objects. In such cases, marshall each object individually. Use a `StringBuilder` class to put the objects in containing elements and output the result as a `String` to the `Response` object. The following is an example marshalling method you could use for this task.

```
public String marshallMe(Object object){
    String xml = "<message>Error</message>";
    try {
      StringWriter st = new StringWriter();

      JAXBContext context = JAXBContext.newInstance(Bid.class,
Auction.class);
      Marshaller marshaller = context.createMarshaller();
      marshaller.setProperty(Marshaller.JAXB_FRAGMENT, true);
      marshaller.marshal(object , st);

      xml = st.toString();
    } catch (Exception e){
      e.printStackTrace();
    }

    return xml;
  }
```


Task 2 - Implement a RESTful Client and Web Forms

Implement a client application that uses the Jersey client API to invoke all the operations provided by the `AuctionManagerRS` web service. In addtional, add HTML files to create auctions and place bids.

Task 3 - Deploy and Test Services in Web Application

Deploy the new `AuctionManagerRS` web service. Test the web service using the auction client application, HTML forms, and browser URLs.

To get the best results, perform the tests in this order.

1. Create an Item. Note the ID assigned.
2. Create a User. Note the ID assigned.
3. Use the IDs from 1 and 2 to create an auction. Note the ID assigned.
4. Create bids on the auction ID created in step 3.
5. With a few bids created, list all the bids and get the highest bid.

Exercise Summary

Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercise.

- Experiences
- Interpretations
- Conclusions
- Applications

Lab 13

Web Services Design Patterns

Objectives

On completion of this lab, you should be able to:

- Decide when to apply web services-based design patterns .

Exercise 1: Understanding Web Services Design Patterns

1. Which pattern has the following design goals:

   - Decouple the input and the output messages.
   - Deliver the output message from the server to the client.
   - Associate an output message to the corresponding input message.

   Choose one:

   a. Asynchronous Interaction

   b. JMS Bridge

   c. Web Service Cache

   d. Web Service Broker

2. Consider a simple design of an application that wants to integrate two remote services into its own workflow. Such a simple design assumes that a transaction begun by the client automatically propagates to both remote services. This allows both remote services to join that same transaction. When they have done so, a failure during processing in the second remote service allows it to roll back its own state by rolling back the current transaction. However, this automatically rolls back the work done by the first remote service, and any work done by the client, during that same transaction.

   Which pattern would you implement in the above scenario to support the transactional behavior of the application? (Choose one).

   a. JMS Bridge

   b. Web Service Cache

   c. Web Service Broker

   d. Web Service Logger

Exercise Summary

Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercise.

- Experiences
- Interpretations
- Conclusions
- Applications

Lab 14

Best Practices and Design Patterns for JAX-WS

There is no exercise for this module.

Lab 15

Best Practices and Design Patterns for JAX-RS

There is no exercise for this module.