Table of Contents

Practices for Lesson 1: Configuring WebLogic and Web Services Tools

Chapter 1

Practices for Lesson 1: Configuring WebLogic and Web Services Tools

Practices Overview

In these practices, you configure your environment to develop Java EE 6 web service applications.

Practice 1-1: Configuring NetBeans to Control WebLogic Server

Overview

In this practice, you add a WebLogic Server instance to the NetBeans IDE. This will enable you to deploy Java EE applications to WebLogic from within NetBeans and also start or stop WebLogic from the Services tab in NetBeans.

Assumptions

NetBeans 7.2 is installed.

Oracle WebLogic Server 12*c* (12.1.1) Zip Distribution is installed.

Tasks

1. Start NetBeans.
2. Click the Tools > Servers menu.
3. Click Add Server.
4. Select Oracle WebLogic Server and click Next.
5. Enter a Server Location of `D:\weblogic\wls\wlserver` and click Next.
6. Verify that the Domain location is `D:/weblogic/domain`. Enter a username and password of `weblogic` and `welcome1`, and click Finish.
7. Click Close.
8. Verify the configuration.
   - Open the Services tab in NetBeans. (Close the Welcome screen if needed.)
   - Expand the Servers node.
   - Right-click Oracle WebLogic Server and click Start.
   - After WebLogic finishes loading, the message “The server started in RUNNING mode.” is displayed in the output window, Right-click Oracle WebLogic Server and click View Admin Console. You should be able to log in with the username and password specified in task 6.

Practice 1-2: Creating and Deploying Web Service Sample Applications

Overview

In this practice, you open and deploy three sample web service–related applications.

Assumptions

WebLogic Server is installed and NetBeans is configured to manage the WLS (WebLogic Server) instance.

Tasks

1. Start NetBeans.
2. Open the CalculatorApp project that contains a SOAP web service.
   - Click the File > Open Project menu item.
   - In the Open Project dialog box, select `D:\labs\student\exercises\lab01\CalculatorApp` and click the Open Project button.
3. Right-click the CalculatorApp project and select Deploy.
   - Deploying a Java EE project should automatically start WebLogic Server if it is not already running. WebLogic Server can also be manually stopped and started by using the Services tab in NetBeans.
   - The CalculatorApp provides a simple SOAP-based web service that can be used to add two integers.
4. Open the Firefox web browser and visit http://localhost:7001/CalculatorApp/CalculatorWSService?wsdl.
   - This URL provides the WSDL file for the calculator web service. A WSDL file is similar to a Java interface; it outlines the operations available in a SOAP web service.
5. Open the CalculatorClientApp project, which functions as a SOAP web service client.
   - Click the File > Open Project menu item.
   - In the Open Project dialog box, select `D:\labs\student\exercises\lab01\CalculatorClientApp` and click the Open Project button.
6. Clean and build the CalculatorClientApp.
   - Java SOAP (JAX-WS) clients depend on machine-generated Java source files. Cleaning and building the project creates these files.
   - Right-click the CalculatorClientApp project and select Clean and Build.
7. Right-click the CalculatorClientApp project and select Run. Use the form to add two numbers.
   - By selecting Run, the project should be deployed and a web browser should open to the URL http://localhost:7001/CalculatorClientApp/. You can also select Deploy and manually launch the web browser.
   - The CalculatorClientApp provides a simple SOAP web service client that can be used to call the web service provided by the CalculatorApp project.
8. Note that the CalculatorClientApp does not add numbers itself. It makes a network call to the CalculatorApp web service. Undeploy the CalculatorApp and try to run the CalculatorClientApp.

- Open the Service tab in NetBeans.
- Expand Servers > Oracle WebLogic Server > Applications > Web Applications.
- Right-click the CalculatorApp application and select Undeploy. This removes the calculator web service from the WebLogic Application Server.
- Return to the Projects tab in NetBeans.
- Attempt to run the CalculatorClientApp project again.
- When you click the Get Result button you should receive an “Error 500--Internal Server Error” message followed by a stack trace.
- This demonstrates a common issue with web service clients—they are networking clients and several things can go wrong either on the remote server or with the network connection between the client and the server. A later lesson discusses error handling.
- Redeploy the CalculatorApp project and verify that the CalculatorClientApp project is functioning again.

9. Open the HelloWorld RESTful web service project.
   - Click the File > Open Project menu item.
   - In the Open Project Dialog select `D:\labs\student\exercises\lab01\HelloWorld` and click the Open Project button.
   - The HelloWorld project implements a REST-style web service and JavaScript client in one project.
10. Open the `index.jsp` web page in the HelloWorld project. You should see a large `<script type="text/javascript">` section that contains JavaScript code.
    - The JavaScript code is the REST web service client. It runs in your web browser when you view the website.
11. Right-click the HelloWorld project and select Run. In the web browser that opens, use the Get and Send hyperlinks to make AJAX calls to the REST web service.
12. Using the FireFox web browser, open the http://localhost:7001/HelloWorld/resources/application.wadl URL. A WADL file is similar to a WSDL file, but for REST web services, it outlines the operations available to clients.

Practice 1-3: Web Service Testing

Overview

In this practice, you try some tools to aid you in testing web services. When developing both the web service client and the web service itself, it can be hard to determine where the problem lies when bugs manifest.

There are many tools available to assist you in the development of web services. Here, you will briefly try out three tools:

- WebLogic Test Client – A SOAP web service testing application that is included in WebLogic Server
- cURL - A command-line utility to make HTTP requests (can be used for both SOAP and REST services)
- FireFox RESTClient Extension – A web browser plug-in that allows you to design custom HTTP requests. Available from https://addons.mozilla.org/en-us/firefox/addon/restclient/.

Assumptions

You have deployed the CalculatorApp and the HelloWorld projects.

Tasks

1. Enable the Favorites tab in NetBeans and add your lab directories for convenient access.
   - In the Window menu, select the Favorites menu item.
   - In the Favorites tab, right-click and select Add to Favorites.
   - In the Add to Favorites dialog box, select the `D:\labs\student\exercises` folder.
   - In the Favorites tab, right-click and select Add to Favorites.
   - In the Add to Favorites dialog box, select the `D:\labs\student\resources` folder.
2. In NetBean's Project tab, select the CalculatorApp.
3. In the CalculatorApp project, expand the Web Services node.
4. Right-click the CalculatorWS node and select Test Web Service.
   - This will activate the WebLogic Test Client web application and point it to the WSDL file for the calculator web service.
   - Use the WebLogic Test Client web page to invoke the calculator service's add operation.
   - After invoking the add operation you can view the add Request Detail summary, which contains the Service Request (SOAP message sent by the client) and the Service Response (SOAP message received from the server).
   - Highlight and copy the Service Request XML to the clipboard.
   - In NetBeans, open the Favorites tab, expand exercises > `lab01`.
   - Right-click the `lab01` folder and create a New > Empty File named `soapreq.xml`.
   - Paste the contents of the clipboard into the new `D:\labs\student\exercises\lab01\soapreq.xml` file and save it.

```
<env:Envelope
xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
  <env:Header />
  <env:Body>
```

```
    <add xmlns="http://calculator.me.org/">
      <i xmlns="">3</i>
      <j xmlns="">1</j>
    </add>
  </env:Body>
</env:Envelope>
```

5. Open a cmd box and change to the `D:\labs\student\resources` directory.

```
D:
cd D:\labs\student\resources
```

6. Use the cURL executable and the SOAP request XML you saved previously to make a SOAP request to the calculator web service.

```
curl --header "Content-type: text/xml" --data
    @D:\labs\student\exercises\lab01\soapreq.xml
    http://localhost:7001/CalculatorApp/CalculatorWSService
```

   - In the response that cURL displays, you should see the total of the add operation. You can try modifying the `soapreq.xml` file's numeric parameters to verify that cURL is obtaining a proper result.
   - Try adding the `-v` option to enable verbose output in the cURL application.

```
curl -v --header "Content-type: text/xml" --data
    @D:\labs\student\exercises\lab01\soapreq.xml
    http://localhost:7001/CalculatorApp/CalculatorWSService
```

   - Though cURL is a low-level utility, it can be useful not only as a testing tool but also as a day-to-day web service client. If you develop shell scripts or batch files, you can use cURL to enable your scripts to act as web service clients.

7. Use the cURL executable to make REST-style requests to the HelloWorld web service.
   - Use cURL to read the current string from the web service.

```
curl --header "Accept: text/plain"
    http://localhost:7001/HelloWorld/resources/helloWorld
```

   - Use cURL to change the string stored by the web service.

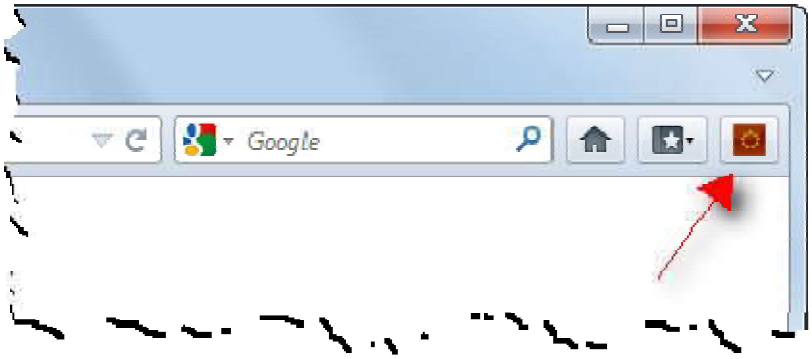
```
curl -X PUT --header "Content-type: text/plain"
    --data "Duke"
    http://localhost:7001/HelloWorld/resources/helloWorld
```

   - Use cURL to read and verify the updated string from the web service.

```
curl --header "Accept: text/plain"
    http://localhost:7001/HelloWorld/resources/helloWorld
```

8. Install the Firefox RESTClient extension.
   - Open the FireFox web browser.
   - Using the Windows File Explorer, navigate to the `D:\labs\student\resources` folder.
   - Drag the `restclient-2.0.3-fx.xpi` file onto the Firefox window. When prompted, click the Install Now and Restart buttons.

9. Start RESTClient extension by clicking the button in the upper-right corner of Firefox.



10. Test the HelloWorld RESTful web service by using the RESTClient extension.
    - Enter http://localhost:7001/HelloWorld/resources/helloWorld into the URL form field and click SEND.

    - View the Response Body (Raw). Notice the format is text/html.

    - Use the Headers menu to add a Custom Header with the values:
        - Name: Accept
        - Value: text/plain
    - Click the SEND button again to re-execute the web service. The response should now be a plain text response.
    - Change the Request method to `PUT` and enter a new string in the Request Body and click SEND.
    - Switch back to the `GET` method and retrieve the updated string.
11. Test the Calculator SOAP web service by using the RESTClient extension. Any tool that can generate a custom HTTP POST request can be used to test a SOAP service.
    - Change the Request Method to POST.
    - Enter http://localhost:7001/CalculatorApp/CalculatorWSService into the URL form field.
    - Remove any custom headers.

- Add a custom header of Content-Type: text/xml.
- Enter a Body that contains a valid SOAP request for the CalculatorWSService.

```
<env:Envelope
xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
  <env:Header />
    <env:Body>
    <add xmlns="http://calculator.me.org/">
      <i xmlns="">3</i>
      <j xmlns="">2</j>
    </add>
  </env:Body>
</env:Envelope>
```

- Click SEND.
- View the response by using the Response Body (Highlight) tab.

Practices for Lesson 2: Creating XML Documents

Chapter 2

Practices for Lesson 2: Creating XML Documents

Practices Overview

In these practices, you create XML documents, XML Schemas, and XML Namespaces.

Practice 2-1: Exploring the PlayingCards Project

Overview

In this practice, you open and explore the PlayingCards project. The PlayingCards project provides a couple of plain old Java object (POJO) classes that can model a collection of playing cards.

Tasks

1. Start NetBeans.
2. Open the PlayingCards project that contains Java classes that model a deck of playing cards.
   - Click the File > Open Project menu item.
   - In the Open Project dialog box, select `D:\labs\student\exercises\lab02\PlayingCards` and click the Open Project button.
3. Explore the PlayingCards application. This is a Java SE project; while running it will display output in the Output window at the bottom of the NetBeans IDE.
   - Open the `App.java` file in the playingcards package and run it. You can run a class with a `main` method by simply right-clicking in the source window for that class and selecting Run File.
   - Beginning with the playingcards.App `main` method, read through the application and explore the source code for all the classes.
   - In the `CardCollection` class find the `show` method. You should see a line that controls which text representation `CardCollection` uses when displaying its playing cards.

     ```
     int displayType = DISPLAY_UNICODE_SUIT;
     ```

   - Try changing the value of `displayType` to each of the following values and running the application.
     - DISPLAY_ABBREVIATION
     - DISPLAY_UNICODE_SUIT
     - DISPLAY_DESCRIPTION
     - DISPLAY_UNICODE6
   - Install the DejaVuSans.ttf font and change the output font so that playing cards render correctly when using DISPLAY_UNICODE6.
     - When using DISPLAY_UNICODE6 the output will most likely resemble square boxes. Java 7 supports the Unicode 6 standard, and Unicode 6 includes a block of playing card symbols, but it is likely that the font used in the output window does not include the playing card symbols.
     - Using Windows File Explorer open the `D:\labs\student\exercises\lab02` folder. Double-click the `dejavu-sans-ttf-2.33.zip` file. Extract the `DejaVuSans.ttf` file from the `ttf` directory to the `D:\labs\student\exercises\lab02` folder.
     - Open the Windows Start menu > Settings > Control Panel. In the control panel, switch to classic view and open Fonts.

- In Fonts, find and delete the existing DejaVu Sans (TrueType) if it is present. Only delete the single DejaVu Sans (TrueType) font; any other fonts should **NOT** be deleted.
- In Fonts, click the File menu and select Install New Font. Using the Add Fonts dialog box, add the `D:\labs\student\exercises\lab02\dejavu-sans-ttf-2.33\ttf\DejaVuSans.ttf` font. This is an updated font that includes the symbols for the playing card Unicode block.
- Restart NetBeans. Run the PlayingCards project again in order to reopen the Output window along the bottom of the NetBeans window.
- Right-click in the NetBeans Output window and select Choose Font. Choose the DejaVu Sans font and a Size of 18 (or greater) and press OK.
- You should now see the Unicode card symbols when running the PlayingCards project with a `displayType` of DISPLAY_UNICODE6.

4. Open the `Card.java` source file from the PlayingCards project and inspect it. In the following practices, you will create an XML document and schema that can be used to represent a playing card.
   - Most of the methods in the `Card` class are used to create the text output of a card in varying formats. Only a small portion the fields or instance variables represent the state of a card.
   - XML documents are typically used to store and transmit data (as opposed to executable code). An XML representation of a card is concerned only with the card's rank, suit, and color.

Practice 2-2: Creating an XML Document

Overview

In this practice, you create an XML document to represent a playing card. After testing the XML document for general conformance to XML structuring rules, you create an XML Schema for a playing card, which will define the tags, attributes, and structure used to represent a playing card.

Assumptions

All applications from the previous practice are still running.

Tasks

1. With NetBeans, create the `cardstack.xml` file in the `src` folder of the PlayingCards project.
   - Right-click the PlayingCards project and select New > Other.
   - In the New File dialog box, select a Category of XML and a File Type of XML Document and click Next.
   - In the New XML Document dialog box, enter a File Name of `cardstack` (the `.xml` extension will be added automatically) and a Folder of `src`. Click Next.
   - In the final dialog box for creating a new file, select a document type of Well-formed Document and click Finish. A well-formed document must follow the basic rules of XML but does not have a dialect (list of allowed tags) defined.
2. Modify the `cardstack.xml` file to contain a stack of three playing cards, one of which should be a joker. Most cards have a rank of 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K, or A, and a suit of CLUBS, DIAMONDS, HEARTS, or SPADES. A Joker is a card that has a rank of JOKER but instead of a suit it has a color of either RED or BLACK.
   - Change the `<root>` tags to `<stack>` tags (opening and closing).
   - The `<stack>` tag should have three child elements of `<card></card>`
   - Cards with a rank of 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K, or A should have child elements of `<rank>` and `<suit>`.
   - Cards with a rank of JOKER should have child elements of `<rank>` and `<color>`. Valid colors are RED and BLACK.

```
<?xml version="1.0" encoding="UTF-8"?>
<stack>
    <card>
        <rank>4</rank>
        <suit>SPADES</suit>
    </card>
    <card>
        <rank>K</rank>
        <suit>CLUBS</suit>
    </card>
    <card>
        <rank>JOKER</rank>
        <color>RED</color>
```

```
    </card>
</stack>
```

3. Check the structure of the `cardstack.xml` file for correctness. All XML documents must be well-formed.
   - Right-click in the `carstack.xml` window and choose Check XML. An XML check window should open at the bottom and show no errors.

```
XML checking started.


Checking
file:/D:/labs/student/exercises/lab02/PlayingCards/src/cardstack
.xml...
XML checking finished.
```

   - Try to introduce errors into the XML file and recheck it to see the types of errors you might possibly encounter.
     - Try to change the closing `</stack>` to `<stack>` (leaving out the forward slash) and recheck. After observing the error in the XML check window, remove the error. Remember, an XML document must have a single root element that encompasses all child elements.
     - Try to swap the last `</color>` and `</card>` tags.

```
    <card>
        <rank>JOKER</rank>
        <color>RED</card>
    </color>
```

     - After observing the error in the XML check window, remove the error. Remember, XML elements must be correctly nested; you cannot begin an element inside a parent and end it after the parent is closed.
     - You should now be comfortable with detecting XML documents for the correct nesting of tag elements.

Practice 2-3: Creating an XML Schema

Overview

In this practice, you create an XML Schema that constrains an XML document to a set of elements that represent a stack of playing cards.

Assumptions

All applications from the previous practice are still running.

Tasks

1. Create the `cardshema.xsd` file in the `src` folder of the PlayingCards project.
   - Right-click the PlayingCards project and select New > Other.
   - In the New File dialog box, select a Category of XML and a File Type of XML Schema (empty) and press Next.
   - In the New XML Schema dialog box, enter a File Name of `cardschema` (the `.xsd` extension will be added automatically) and a Folder of `src`. Press Finish.
2. When creating an XML Schema it will be easier to create and manage the schema if you take a modular approach. Just as you do not (usually) design Java applications to use a single class, you do not (usually) see XML Schemas with a single global named type.
3. Unless instructed otherwise, all types you add to the schema should be inside of the `xs:schema` element. Remember, an XML Schema is an XML document itself so it can have only one top-level element.
4. Create a `simpleType` named `suitType` that only allows the strings CLUBS, DIAMONDS, HEARTS, or SPADES.
   - When used in an XML document, a `suitType` example might be:

     ```
     <suit>SPADES</suit>
     ```

   - `suitType` is the XML schema equivalent of the Java `playingcards.Suit enum`.
   - `suiteType` should be created by using a restriction of `xs:string`.
   - Use an enumeration to limit the string values to the same suits allowed by the Suit Java enum.

     ```
     <xs:simpleType name="suitType">
         <xs:restriction base="xs:string">
             <xs:enumeration value="CLUBS" />
             <xs:enumeration value="DIAMONDS" />
             <xs:enumeration value="HEARTS" />
             <xs:enumeration value="SPADES" />
         </xs:restriction>
     </xs:simpleType>
     ```

5. Create a `simpleType` named `colorType` that allows only the strings RED or BLACK.
   - When used in an XML document, a `colorType` example might be:

     ```
     <color>RED</color>
     ```

   - `colorType` should be created by using a restriction of `xs:string`.
   - Use an enumeration to limit the string values to RED or BLACK.

```
<xs:simpleType name="colorType">
    <xs:restriction base="xs:string">
        <xs:enumeration value="RED" />
        <xs:enumeration value="BLACK" />
    </xs:restriction>
</xs:simpleType>
```

6. Create a `simpleType` named `rankType` that allows only the strings 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K, A, or JOKER.
   - When used in an XML document, a `rankType` example might be:

```
<rank>4</rank>
```

   - `rankType` is the XML Schema equivalent of the Java `playingcards.Rank enum`.
   - `rankType` should be created by using a restriction of `xs:string`.
   - Use an enumeration to limit the string values to the same ranks allowed by the Rank Java enum.

```
<xs:simpleType name="rankType">
    <xs:restriction base="xs:string">
        <xs:enumeration value="2" />
        <xs:enumeration value="3" />
        <xs:enumeration value="4" />
        <xs:enumeration value="5" />
        <xs:enumeration value="6" />
        <xs:enumeration value="7" />
        <xs:enumeration value="8" />
        <xs:enumeration value="9" />
        <xs:enumeration value="10" />
        <xs:enumeration value="J" />
        <xs:enumeration value="Q" />
        <xs:enumeration value="K" />
        <xs:enumeration value="A" />
        <xs:enumeration value="JOKER" />
    </xs:restriction>
</xs:simpleType>
```

7. Create a `complexType` named `cardType` to represent a single playing card.
   - When used in an XML document, a `cardType` example might be:

```
<card>
    <rank>4</rank>
    <suit>SPADES</suit>
</card>
```

   - Another `rankType` example might be:

```
<card>
    <rank>JOKER</rank>
```

```
    <color>RED</color>
</card>
```

- Notice that a card can have a suit or color but not both.
- `cardType` is the XML schema equivalent of the Java `playingcards.Card` class.
- Because the `cardType` must have nested elements (rank, suit, color) it must be created by using `complexContent`. The use of `complexContent` means `cardType` must be of a `complexType`. Create a `complexType` named `cardType`.
- Inside the `complexType` named `cardType`, create a `complextContent` element.
- Inside the `complextContent` element, create a restriction with a base type of `xs:anyType`.
- Inside the restriction element, create a sequence that will list the elements allowed in a card element.
- The first element allowed in the sequence of elements inside a card should be named `rank`. The "name" here defines the name used in XML documents that use this schema. The "rank" element should be of type `rankType` which you defined in step 6.
- After the rank, a card can have either a suit or a color. In an XML Schema, this is a "choice". Create a choice after the rank element.
- The choice should be between two elements. One is an element named `suit` of type `suitType` (from step 4) and the other should be named `color` and be of type `colorType` (from step 5).
- The completed `complexType` named `cardType` should resemble the following example.

```
<xs:complexType name="cardType">
    <xs:complexContent>
        <xs:restriction base="xs:anyType">
            <xs:sequence>
                <xs:element name="rank" type="rankType" />
                <xs:choice>
                    <xs:element name="suit" type="suitType" />
                    <xs:element name="color" type="colorType" />
                </xs:choice>
            </xs:sequence>
        </xs:restriction>
    </xs:complexContent>
</xs:complexType>
```

8. Create a `complexType` named `stackType` to represent a stack of cards.
   - When used in an XML document, a `stackType` example might be:

```
<stack>
    <card>
        <rank>4</rank>
        <suit>SPADES</suit>
    </card>
    <card>
```

```
        <rank>2</rank>
        <suit>CLUBS</suit>
    </card>
    <card>
        <rank>JOKER</rank>
        <color>RED</color>
    </card>
</stack>
```

- Create a `complexType` named `stackType`.
- Because a `stackType` contains elements (cards) it is a `complexType` with `complexContent` that is based on a restriction of the `xs:anyType` base type. Remember that a `complexType` defaults to having `complexContent` with a restriction using the `xs:anyType` base type. Use the default content type and restriction (by not specifying one).
- Add a sequence to the `stackType`.
- A card stack is just a sequence of an unlimited number of card elements. Add an element named `card` of type `cardType` to the sequence. The maximum number of card occurrences should be unbounded.
- The completed `stackType` should resemble the following example:

```
<xs:complexType name="stackType">
    <xs:sequence>
        <xs:element name="card" type="cardType"
maxOccurs="unbounded" />
    </xs:sequence>
</xs:complexType>
```

9. The `stackType complexType` should be allowed as an element named `stack` in XML documents.
   - Add an element named `stack` with a type of `stackType` (created in step 8).

```
<xs:element name="stack" type="stackType" />
```

10. Constrain the `cardstack.xml` document you created in Practice 2-2 by using the XML Schema you just created.
    - Open the `cardstack.xml` document.
    - Add the XMLSchema-instance namespace to the `stack` element.

```
<stack xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
```

    - Use the `xsi:noNamespaceShemaLocation` attribute to point any parsers that read the `cardstack.xml` file to the `cardschema.xsd` file.

```
<stack xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="cardschema.xsd">
```

    - The `noNamespaceSchemaLocation` attribute is used when you have a schema that does not declare a target namespace. You will add namespaces in a subsequent practice.
11. Validate the `cardstack.xml` document by using the `cardschema.xsd` XML Schema.

- In the `cardstack.xml` document, right-click and choose Validate XML. The Validate XML option checks that an XML document conforms to its schemas whereas the Check XML option only checks that a document is well formed. When validating, the output should be:

```
XML validation started.


Checking
file:/D:/labs/student/exercises/lab02/PlayingCards/src/cardstack
.xml...
Referenced entity at
"file:/D:/labs/student/exercises/lab02/PlayingCards/src/cardsche
ma.xsd".
XML validation finished.
```

12. Modify the `cardstack.xml` document to violate the `cardschema.xsd` XML schema.
    - Keep the `cardstack.xml` file well-formed.
    - Try modifying the content of elements to contain content that is not allowed by the schema. Possible errors you can introduce include:
      - Elements that are not allowed. Example: add a `<person></person>` to the stack.
      - Values that are not allowed. Example: Change an element to have a value that is not allowed.

```
<card>
    <rank>JOKER</rank>
    <color>BLUE</color>
</card>
```

    - Use the Check XML action to verify the XML document is still well-formed (it should be).
    - Use the Validate XML action to verify the XML document violates its schema. The output should resemble:

```
XML validation started.


Checking
file:/D:/labs/student/exercises/lab02/PlayingCards/src/cardstack
.xml...
Referenced entity at
"file:/D:/labs/student/exercises/lab02/PlayingCards/src/cardsche
ma.xsd".
cvc-enumeration-valid: Value 'BLUE' is not facet-valid with
respect to enumeration '[RED, BLACK]'. It must be a value from
the enumeration. [14]
cvc-type.3.1.3: The value 'BLUE' of element 'color' is not
valid. [14]
XML validation finished.
```

    - After testing your schema by introducing several changes, you should repair the `cardstack.xml` file. Verify that the document passes validation before continuing.
13. Compare your XML Schema that uses global types to an equivalent one that uses nested types.

- Using the Favorites tab in NetBeans, copy the `D:\labs\student\exercises\lab02\nestedtypescardschema.xsd` file to your project and open the copy in your project. The copy should be placed in the same location as the `cardschema.xsd` file.
- Notice how everything is nested in the `nestedtypescardschema.xsd` file.
- Notice how all the `complexType` and `simpleType` elements no longer have declared names.
- When you create a `complexType` or a `simpleType` as a direct child of `<xs:schema>` and assign them names, you are creating global types. Global types can be reused and break up schema declarations into smaller chucks.
- When you create a `complexType` or a `simpleType` as the child of an `<xs:element>`, that nested type is the type of the parent `<xs:element>`. These embedded types are known as nested types.
- Nested types are not bad but should be eliminated if you find yourself copying and pasting complex or simple type declarations multiple times, or if you find yourself having a hard time understanding what the corresponding validated XML should look like. Many schema documents use a combination of global and nested type declarations.
- Verify that the `nestedtypescardschema.xsd` file is equivalent to the `cardschema.xsd` by modifying the `cardstack.xml` document to use the `nestedtypescardschema.xsd` file.

```
<stack xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="nestedtypescardschema.xsd">
```

- Repeat step 12 to test the `nestedtypescardschema.xsd` file. After repeating step 12 switch the `cardstack.xml` document back to using the `cardschema.xsd` file.

```
<stack xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="cardschema.xsd">
```

Practice 2-4: Using XML Namespaces

Overview

In this practice, you modify your XML documents and Schemas to use XML Namespaces.

Assumptions

All applications from the previous practice are still running.

Tasks

1. Add the `games.xsd` schema to the PlayingCards project.
   - The `games.xsd` schema defines a dialect that can be used to represent games of Indian Rummy (also called 13 Card Rummy).
   - Using the Favorites tab in NetBeans, copy the `D:\labs\student\exercises\lab02\games.xsd` file to your project and open the copy in your project. The copy should be placed in the same location as the `cardschema.xsd` file.
   - Note the `<xs:include>` that is used to make the types defined in the `cardschema.xsd` file available throughout the `games.xsd` file. At this point, neither XSD file belongs to a declared namespace and, therefore, they belong to the default namespace. `<xs:include>` is used to effectively combine schemas that belong to the same namespace.

```
<xs:include schemaLocation="cardschema.xsd"/>
```

2. Add the `games.xml` file to the PlayingCards project.
   - The `games.xml` file can be used to store the state of multiple `<indian-rummy>` games. The hands, stock piles, and discard piles are kept small for demonstration purposes.
   - Using the Favorites tab in NetBeans, copy the `D:\labs\student\exercises\lab02\games.xml` file to your project and open the copy in your project. The copy should be placed in the same location as the `games.xsd` file.
   - Inspect the `games.xml` file. Notice that it only references one schema location directly, yet still uses card elements. This is because the `games.xsd` schema includes the `cardschema.xsd`.

```
<games xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="games.xsd">
```

3. Currently both XML Schemas belong to the default namespace. It is a good practice (and sometimes required) that XML Schemas belong to different namespaces.
   - Look at both schema files; neither should have a `targetNamespace` attribute in the `<xs:schema>` element.
   - Namespaces provide a way for multiple schema vendors to avoid naming conflicts. Just like there are multiple `Date` classes in the Java libraries, you may encounter multiple elements of the same name that are used for different purposes. In the `games.xml` file there are two `<rank>` elements, one for how a player is ranked and also for the rank of a card. You can think of XML Namespaces as being similar to Java packages.

4. Assign the components in `cardschema.xsd` to the `urn:dukesdecks` namespace and make it the default namespace.
   - A manufacturer of playing cards might define the XML dialect that can be used to represent a playing card or a collection of playing cards, but because playing cards can be used for multiple types and variations of games, they would not be concerned with the dialect for each game.
   - Open the `cardschema.xsd` file and add a `targetNamespace="urn:dukesdecks"` attribute to `<xs:schema>`. This causes all types and elements created in the `cardschema.xsd` file to reside in the `urn:dukesdecks` namespace.
   - Add an `xmlns="urn:dukesdecks"` attribute to `<xs:schema>`.

```
<xs:schema version="1.0"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns="urn:dukesdecks"
           elementFormDefault="qualified"
           targetNamespace="urn:dukesdecks">
```

   - The purpose of setting `urn:dukesdecks` as the default namespace is to eliminate the need to qualify the types when they are used. Given this line from `cardschema.xsd` notice that there is no prefix before `stackType`, without a default namespace you would be required to add a prefix.

```
<xs:element name="stack" type="stackType" />
```

5. Modify `cardstack.xml` to use the `urn:dukesdecks` namespace.
   - Open the `cardstack.xml` file.
   - Modify the stack root element.
     - Replace the `noNamespaceSchemaLocation` attribute with:

```
xsi:schemaLocation="urn:dukesdecks cardschema.xsd"
```

     - The `schemaLocation` attribute takes multiple pairs of strings separated by white space. The first part of the pair identifies a namespace and the second part of the pair specifies where that namespace in defined.
     - Add an `xmlns` attribute to map the `dd` prefix to the `urn:dukesdecks` namespace.

```
xmlns:dd="urn:dukesdecks"
```

     - Modify all tags to use the `dd` prefix. The beginning of the file should resemble:

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

<dd:stack xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:dd="urn:dukesdecks"
        xsi:schemaLocation="urn:dukesdecks cardschema.xsd">
    <dd:card>
        <dd:rank>4</dd:rank>
        <dd:suit>SPADES</dd:suit>
    </dd:card>
```

   - Validate the `cardstack.xml` file. Fix any errors that may appear in either `cardstack.xml` or `cardschema.xsd`.

6. Assign the components in `games.xsd` to the `urn:games` namespace without making it the default namespace.
   - Open the `games.xsd` file and add a `targetNamespace="urn:games"` attribute to `<xs:schema>`. This causes all types and elements created in the `games.xsd` file to reside in the `urn:games` namespace.
   - Add an `xmlns:dd="urn:dukesdecks"` attribute to `<xs:schema>`.
   - Add an `xmlns:cg="urn:games"` attribute to `<xs:schema>`.

```
<xs:schema version="1.0"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:dd="urn:dukesdecks"
           xmlns:cg="urn:games"
           elementFormDefault="qualified"
           targetNamespace="urn:games">
```

   - Because the `cardschema.xsd` and `games.xsd` element are now in different namespaces, you must switch the `<xs:include>` to a `<xs:import>`.

```
<xs:import namespace="urn:dukesdecks"
schemaLocation="cardschema.xsd" />
```

   - Because there is no default namespace in `games.xsd` and because you are importing elements from a different namespace, you must use the declared `dd` and `cg` prefixes to qualify all `type` usage. For example, `type="stackType"` becomes `type="dd:stackType"`.
   - Remember the creation of an element within a namespace does not need a prefix, just a reference to that type (`ref` and `type` attributes primarily). This snippet remains unchanged `<xs:complexType name="gamesType">` although wherever you referenced `gamesType` it must be updated to `<xs:element name="games" type="cg:gamesType" />`.
   - Make sure all `type= "... "` attributes use a prefix in the `games.xsd` file.
7. Modify `games.xml` to use the `urn:games` and `urn:dukesdecks` namespaces.
   - Open the `games.xml` file.
   - Modify the stack root element.
     - Replace the `noNamespaceSchemaLocation` attribute with

```
xsi:schemaLocation="urn:games games.xsd".
```

     - Add an `xmlns` attribute to map assign `urn:games` as the default namespace.

```
xmlns="urn:games"
```

     - Add an `xmlns` attribute to map the `dd` prefix to the `urn:dukesdecks` namespace.

```
xmlns:dd="urn:dukesdecks"
```

     - The beginning of the file should now resemble:

```
<games xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns="urn:games"
       xmlns:dd="urn:dukesdecks"
       xsi:schemaLocation="urn:games games.xsd">
```

- Because `urn:games` is the default namespace, the `games` element and any other elements that are a part of `urn:games` do not require a prefix.
- However, any `urn:dukesdesks` elements must be qualified with the `dd` prefix. Modify all the card-related elements to have the required prefix. After finishing, a player's hand should now resemble:

```
<hand>
    <dd:card>
        <dd:rank>7</dd:rank>
        <dd:suit>SPADES</dd:suit>
    </dd:card>
    <dd:card>
        <dd:rank>3</dd:rank>
        <dd:suit>CLUBS</dd:suit>
    </dd:card>
</hand>
```

- Pay special attention to any `<rank>` tags. The rank of a player is part of the `urn:games` namespace, (which you have configured as the default for the `games.xml` document) whereas the rank of a card is part of the `urn:dukesdecks` namespace and must be qualified.
- Validate the `games.xml` file. Fix any errors that occur.

Practices for Lesson 3: Processing XML with JAXB

Chapter 3

Practices for Lesson 3: Processing XML with JAXB

Practices Overview

In these practices, you produce and consume XML documents by using JAXB.

Practice 3-1: Creating Java Classes from XML Schemas

Overview

In this practice, you use an existing XML Schema to generate Java classes that you then use to read and write XML documents.

Tasks

1. Start NetBeans.
2. Create a new Java Application project named SchemaToJava in the `D:\labs\student\exercises\lab03` folder.
   - From the NetBeans menu choose File > New Project.
   - Choose a Category of Java.
   - Choose a Project Type of Java Application.
   - Click Next.
   - Change the Project Name to SchemaToJava.
   - Change the Project Location to `D:\labs\student\exercises\lab03`.
   - Press Finish.
3. Create a new JAXB Binding for the `cardschema.xsd` you used in Practice 2-4.
   - Right-click the SchemaToJava project and choose New > Other.
   - Choose a Category of XML.
   - Choose a File Type of JAXB Binding.
   - Specify a Binding Name of CardBinding.
   - Specify a Schema File Location of `../../lab02/PlayingCards/src/cardschema.xsd`.
   - Specify a Package Name of jaxbcards.
   - Click Finish.
4. Generate the JAXB classes.
   - The xjc Ant task does not run when you execute the project or run a file in it. There are two ways you can execute the xjc task to generate the JAXB classes.
     - Right-click the SchemaToJava project and choose Build.
     - Right-click the JAXB Bindings node under the SchemaToJava project and choose Regenerate Java Code.
5. Explore the JAXB generated classes.
   - A NetBeans JAXB Binding configures an xjc Ant task to run as part of the project build.
   - The SchemaToJava project should contain a new Generated Sources branch in the project view. There you will find:
     - `package-info.java` – Used to add annotation at the package level
     - `ObjectFactory.java` – A factory used to create instances of objects that represent XML elements
     - Several `*Type.java` classes – Each Type class corresponds to a type defined in the parsed XML schema
6. Read an XML document that conforms to the `cardschema.xsd` file.

- Copy the `cardstack.xml` file from the PlayingCards project to the Source Packages folder in the SchemaToJava project.
- Open the `SchemaToJava.java` file in the SchemaToJava project.
- In the `main` method, create a `try` block to handle the two types of exceptions that will possibly be thrown by the code you are about to add:

```
try {

} catch (JAXBException | FileNotFoundException ex) {
    ex.printStackTrace();
}
```

  - Add any required imports (you can run the Fix Imports tool by right-clicking in the source window).
- In the `try` block, create a new JAXBContext by using the `jaxbcards` package.

```
JAXBContext jc = JAXBContext.newInstance("jaxbcards");
```

- Create an Unmarshaller.

```
Unmarshaller u = jc.createUnmarshaller();
```

- Create an InputStream to read from the `cardstack.xml` file.

```
InputStream in = new FileInputStream("src/cardstack.xml");
```

- Unmarshall the root element of the `cardstack.xml` file.

```
JAXBElement<StackType> rootElement =
(JAXBElement<StackType>)u.unmarshal(in);
StackType cs = rootElement.getValue();
```

- Iterate through the cards and print out their values.

```
List<CardType> cards = cs.getCard();
for (CardType card : cards) {
    String rank = card.getRank();
    if(rank.equalsIgnoreCase("joker")) {
        System.out.println(card.getColor() + " " + rank);
    } else {
        System.out.println(rank + " of " + card.getSuit());
    }
}
```

7. Open the PlayingCards project (if it is not still open), which contains Java classes that model a deck of playing cards.
   - Click the File > Open Project menu item.
   - In the Open Project dialog box, select `D:\labs\student\exercises\lab02\PlayingCards` and click the Open Project button.
8. Run the SchemaToJava project and correct any errors that occur.

```
run:
4 of SPADES
```

```
2 of CLUBS
RED JOKER
BUILD SUCCESSFUL (total time: 0 seconds)
```

Note: If you run the file instead of the project, you may receive errors indicating that the jaxbcards package and its classes cannot be found.

Practice 3-2: Creating XML Schemas from JAXB Annotated Classes

Overview

In this practice, you use an existing XML Schema to generate Java classes, which you then use to read and write XML documents.

Tasks

1. Start NetBeans.
2. Create a new Java Application project named JavaToSchema in the `D:\labs\student\exercises\lab03` folder.
   - From the NetBeans menu choose File > New Project.
   - Choose a Category of Java.
   - Choose a Project type of Java Application.
   - Click Next.
   - Change the Project Name to JavaToSchema.
   - Change the Project Location to `D:\labs\student\exercises\lab03`.
   - Click Finish.
3. Open the PlayingCards project from Practice 2-4 (if it is not already open).
   - From the NetBeans menu choose File > Open Project.
   - Select `D:\labs\student\exercises\lab02\PlayingCards`.
   - Click the Open Project button.
4. Copy the playingcards package from the PlayingCards project to the JavaToSchema project.
5. In the JavaToSchema project, delete the `App.java` file in the playingcards package.
6. Modify the remaining classes in the playingcards package of the JavaToSchema project to enable their use as JAXB classes. Most of the changes relate to the introduction of annotations. The goal is to make the schema generated from these classes resemble the `cardschema.xsd` file from the PlayingCards project.
7. In the JavaToSchema project, add JAXB annotations to the `CardCollection` class.
   - At the class level add an `@XmlAccessorType` annotation specifying that only fields should be marshaled.
   - At the class level, add an `@XmllType` annotation that causes the generated schema to name the `complexType` for this class as `stackType`. Also specify a `propOrder` attribute with a single value of `cards`.
   - At the class level, add an `@XmlRootElement` that indicates the element name is `stack` and the namespace is `urn:dukesdecks`.

     ```
     @XmlAccessorType(XmlAccessType.FIELD)
     @XmlType(name = "stackType", propOrder = {
         "cards"
     })
     @XmlRootElement(name="stack", namespace="urn:dukesdecks")
     ```

   - On the `cards` field, add an `@XmlElement` that sets the element name to `card` (instead of defaulting to the name of the field) and make the element required; otherwise, the resulting schema element will have `minOccurs= "0"`.

```
@XmlElement(name = "card", required = true)
```

- Add any required imports.

8. In the JavaToSchema project, add JAXB annotations to the Suit enum.
   - At the class level, add an `@XmlEnum` annotation. By default, the name of each enumerated instance in the Java enum will be used as a value in the resulting schema.
   - At the class level, add an `@XmllType` annotation that causes the generated schema to name the `complexType` for this class to be `suitType`.

```
@XmlType(name = "suitType")
@XmlEnum
```

   - Add any required imports.

9. In the JavaToSchema project, add JAXB annotations to the Rank enum.
   - At the class level, add an `@XmlEnum` annotation.
   - At the class level, add an `@XmllType` annotation that causes the generated schema to name the `complexType` for this class as `rankType`.

```
@XmlType(name = "rankType")
@XmlEnum
```

   - Use an `@XmlEnumValue` annotation on each field to specify the values allowed by the schema. The allowed values should be A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K, JOKER. Example:

```
@XmlEnumValue("K")
KING("King", "K"),
@XmlEnumValue("JOKER")
JOKER("Joker", "Joker");
```

   - Add any required imports.

10. In the JavaToSchema project, add a new Color enum to the playingcards package and annotate it with JAXB annotations.
    - Currently the `Card` class uses the `java.awt.Color` class as the type for a color value. Because this class is built in to the Java API, you do not have an opportunity to add JAXB annotations to it; therefore, you must replace its use with a new class or enum.
    - Create a new enum in the playingcards package named Color with the fields WHITE, RED, and BLACK.
    - At the class level, add an `@XmlEnum` annotation.
    - At the class level, add an `@XmllType` annotation that causes the generated schema to name the `complexType` for this class as `colorType`.

```
@XmlType(name = "colorType")
@XmlEnum
public enum Color {
    WHITE,
    RED,
    BLACK;
}
```

11. In the JavaToSchema project, add JAXB annotations to the `Card` class.

- Remove the import of java.awt.Color; you will now use the Color enum created in the previous step.
- At the class level, add an `@XmlAccessorType` annotation with a value of `XmlAccessType.NONE`. Only elements with JAXB annotations will be marshaled to XML.
- At the class level, add an `@XmllType` annotation that causes the generated schema to name the `complexType` for this class as `cardType` and use a `propOrder` of `rank`, then `suitOrColor`.

```
@XmlAccessorType(XmlAccessType.NONE)
@XmlType(name = "cardType", propOrder = {
    "rank",
    "suitOrColor"
})
```

- Make the `rank` filed required.

```
@XmlElement(required = true)
private Rank rank;
```

- The `suitOrColor` property does not exist yet. The `Card` class has three fields: rank, suit, and color and, whereas rank should always be marshaled, you only ever want either color or suit. This corresponds to a `<xs:choice>` in the XML Schema.
- Add the following code to the `Card` class to represent a `<xs:choice>`.

```
@XmlElements(value = {
    @XmlElement(name = "suit",
    type = Suit.class,
    required=true),
    @XmlElement(name = "color",
    type = Color.class,
    required=true)
})
private Object getSuitOrColor() {
    if (rank != null && rank == Rank.JOKER) {
        return color;
    } else {
        return suit;
    }
}

private void setSuitOrColor(Object object) {
    if (object instanceof Color) {
        color = (Color)object;
    } else if (object instanceof Suit) {
        suit = (Suit)object;
    }
```

```
}
```

- Add any required imports.

12. Create a new Java Package Info file in the playingcards package.
    - Right-click the playingcards package and choose New > Other.
    - Choose the Java category and a File Type of Java Package Info, click Next and then Finish.
    - Add a `@javax.xml.bind.annotation.XmlSchema` annotation to the package, which sets the schema's targetNamespace to `"urn:dukesdecks"` and sets the `elementFormDefault` attribute to "`qualified`".

```
@javax.xml.bind.annotation.XmlSchema(
        namespace = "urn:dukesdecks",
        elementFormDefault =
javax.xml.bind.annotation.XmlNsForm.QUALIFIED)
```

13. Create a `jaxb.index` file in the playingcards package.
    - A `jaxb.index` file is an easy way to allow you to initialize a JAXBContext using a package name. In an upcoming step, you will initialize a JAXBContext as follows:

```
JAXBContext context = JAXBContext.newInstance("playingcards");
```

    - Right-click the playingcards package and choose New > Other.
    - Choose the Other category and a File Type of Empty File and click Next. Name the file `jaxb.index` and click Finish.
    - Open the `jaxb.index` file and add the short name of each JAXB annotated class file.

```
Card
CardCollection
Color
Rank
Suit
```

14. Modify the `main` method in the `JavaToSchema.java` file to programmatically generate an XML Schema file.
    - Create a JAXBContext object by using the playingcards package name.
    - Use the `generateSchema` method in the JAXBObject to generate an XML Schema. The schema should be saved in the `src/cardschema.xsd` file.

```
try {
    JAXBContext context =
JAXBContext.newInstance("playingcards");
    context.generateSchema(new SchemaOutputResolver() {
        @Override
        public Result createOutput(String namespaceUri, String
fileName) throws IOException {
            return new StreamResult(new
File("src/cardschema.xsd"));
        }
    });
```

```
} catch (JAXBException | IOException ex) {
    ex.printStackTrace();
}
```

- Add any required imports.
- Save all your changes and run the JavaToSchema project. A `cardschema.xsd` file should appear in the `<default package>` branch.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema elementFormDefault="qualified" version="1.0"
targetNamespace="urn:dukesdecks" xmlns:tns="urn:dukesdecks"
xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xs:element name="stack" type="tns:stackType"/>

    <xs:complexType name="cardType">
        <xs:sequence>
            <xs:element name="rank" type="tns:rankType"/>
            <xs:choice>
                <xs:element name="suit" type="tns:suitType"/>
                <xs:element name="color" type="tns:colorType"/>
            </xs:choice>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="stackType">
        <xs:sequence>
            <xs:element name="card" type="tns:cardType"
maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>

    <xs:simpleType name="rankType">
        <xs:restriction base="xs:string">
            <xs:enumeration value="A"/>
            <xs:enumeration value="2"/>
            <xs:enumeration value="3"/>
            <xs:enumeration value="4"/>
            <xs:enumeration value="5"/>
            <xs:enumeration value="6"/>
            <xs:enumeration value="7"/>
            <xs:enumeration value="8"/>
            <xs:enumeration value="9"/>
            <xs:enumeration value="10"/>
```

```
            <xs:enumeration value="J"/>
            <xs:enumeration value="Q"/>
            <xs:enumeration value="K"/>
            <xs:enumeration value="JOKER"/>
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="suitType">
        <xs:restriction base="xs:string">
            <xs:enumeration value="CLUBS"/>
            <xs:enumeration value="DIAMONDS"/>
            <xs:enumeration value="HEARTS"/>
            <xs:enumeration value="SPADES"/>
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="colorType">
        <xs:restriction base="xs:string">
            <xs:enumeration value="WHITE"/>
            <xs:enumeration value="RED"/>
            <xs:enumeration value="BLACK"/>
        </xs:restriction>
    </xs:simpleType>
</xs:schema>
```

15. Continue modifying the `JavaToSchema` class by adding the code to create a deck of playing cards, shuffle them, and save them to the `src/cardstack.xml` file.
    - At the bottom of the `try` block created in the previous step, create a new FrenchCardDeck and shuffle it.

```
CardCollection deck = new FrenchCardDeck();
deck.shuffle();
```

    - Create a Marshaller object that will convert your JAXB annotated classes to XML.

```
Marshaller m = context.createMarshaller();
```

    - Configure the Marshaller to produce pretty output and include `xsi:schemaLocation` in the resulting XML document.

```
m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
m.setProperty(Marshaller.JAXB_SCHEMA_LOCATION,
"cardschema.xsd");
```

    - Marshall the FrenchCardDeck to the `src/cardstack.xml` file.

```
m.marshal(deck, new File("src/cardstack.xml"));
```

    - Add any required imports.
    - Run the project. Inspect the resulting `cardstack.xml`, which should appear in the `<default package>`. The XML document should resemble the following output:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<stack xmlns="urn:dukesdecks"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="cardschema.xsd">
    <card>
        <rank>2</rank>
        <suit>HEARTS</suit>
    </card>
    <card>
        <rank>K</rank>
        <suit>SPADES</suit>
    </card>
    <!-- more cards not shown -->
</stack>
```

(Optional) Practice 3-3: Creating Java Classes from XML Schemas

Overview

This practice is OPTIONAL. Please verify with your instructor that you have time to complete it.

In this practice, you improve the creation of the xjb generated class by adding a custom JAXB binding instruction.

Tasks

1. Open the `D:\labs\student\exercises\lab03\SchemaToJava` project from Practice 3-1.
2. There are several issues with the xjc generated classes.
   - The class names may not match what you want. CardType is generated and you might want just Card.
   - The method names may not match what you want. The `StackType` method `getCard()` returns a `List<CardType>`. Java convention would use a method called `getCards()`.
   - There is no Rank enum even though, in the XML Schema, the `rankType` uses only enumerated values. If you look at the `CardType` class, the rank is represented by a Java String.

   Many issues like these can be corrected by using custom JAXB bindings. Custom bindings can be accomplished inline (in the XML Schema file if it can be modified) or with an external binding file.
3. Modify `cardschema.xsd` to support inline custom bindings.
   - Open the `cardschema.xsd` file under the JAXB Bindings branch of the project.
   - Do NOT right-click this file and choose refresh for the remainder of this practice. Refreshing the file will replace the `cardschema.xsd` in this project with a copy from its original location. You can determine the location of the original version of `cardschema.xsd` by right-clicking the CardBinding node and choosing Change JAXB Options.
   - Add the required attributes to `<xs:schema>` to support JAXB custom bindings.

```
<xs:schema version="1.0"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
           xmlns="urn:dukesdecks"
           elementFormDefault="qualified"
           targetNamespace="urn:dukesdecks"
           jaxb:version="2.1">
```

4. Use inline binding customization to modify the `StackType` generated class to have a `getCards()` method instead of `getCard()`.
   - Find the `stackType complexType` in the `cardschema.xsd`.
   - Modify the `stackType` by adding a `<jaxb:property>` element, which sets the card element to have a JAXB property name of `cards`.

```
<xs:complexType name="stackType">
    <xs:sequence>
```

```
        <xs:element name="card" type="cardType"
maxOccurs="unbounded">
            <xs:annotation>
                <xs:appinfo>
                    <jaxb:property name="cards"/>
                </xs:appinfo>
            </xs:annotation>
        </xs:element>
    </xs:sequence>
</xs:complexType>
```

- Right-click the JAXB Bindings branch and choose Regenerate Java Code.
- The `SchemaToJava.java` file should now have an error. Correct the error by changing the call of `getCard()` to `getCards()`.
- Run the project.

5. Use inline binding customization to generate a Rank enumeration.
   - Currently the `rankType simpleType` is not being translated to a Java enum by xjc. This is because some of the enumerated values start with a number.

```
<xs:enumeration value="2" />
```

The above line is a source of difficulty because Java identifiers cannot begin with a number.

```
public enum RankType {
    2; // illegal
}
```

   - To correct the issue, find the `rankType simpleType` and modify each enumerated value. The line:

```
<xs:enumeration value="2" />
```

Would change to:

```
<xs:enumeration value="2">
     <xs:annotation>
          <xs:appinfo>
               <jaxb:typesafeEnumMember name="TWO"/>
          </xs:appinfo>
     </xs:annotation>
 </xs:enumeration>
```

   - Right-click the JAXB Binding branch and regenerate the Java code. You should now see a `RankType` class.
   - Modify the `SchemaToJava` class to correctly use the RankType enum.

```
for (CardType card : cards) {
    RankType rank = card.getRank();
    if (rank == RankType.JOKER) {
        System.out.println(card.getColor() + " " + rank);
    } else {
```

```
            System.out.println(rank + " of " + card.getSuit());
        }
}
```

- Run the project.

Now that you have seen inline customizations, you will learn how to accomplish the same thing by using an external binding file.

6. Discard all your inline changes.
   - Right-click the cardschema.xsd node in the project tree and click Refresh. This action should recopy the `cardschema.xsd` file from Practice 2-4 into the SchemaToJava project and delete all your inline customizations.
   - You should have compiler errors in `SchemaToJava.java`; leave them in place.
7. Add an external custom binding file to the JAXB configuration.
   - Right-click the CardBinding node of the project and choose Change JAXB Options.
   - Select the check box that says Use Binding File.
   - Click the Configure button to the right of the Use Binding File check box.
   - Click the Add button and add the `D:\labs\student\exercises\lab03\bindings.xjb.xml` file. Click OK.
8. Regenerate the Java code and run the project.
   - Right-click the JAXB Bindings branch and choose Regenerate Java Code.
   - Run the project.
9. Examine the `bindings.xjb.xml` file.
   - An external binding customization file is a means to perform the same change as inline customization when you cannot or do not want to modify your schema file.
   - The file begins with the location of the schema to which the customizations apply.

```
<jaxb:bindings schemaLocation="cardschema.xsd">
```

   - Inline customizations are context aware. The placement of inline customizations determines which node they apply to. Because external customization are not in the context of the nodes, extra information must be added to specify the node being customized.

```
<jaxb:bindings
node="//xs:complexType[@name='stackType']/xs:sequence/xs:element
[@name='card']">
    <jaxb:property name="cards"/>
</jaxb:bindings>
```

   - In the example above, the `node` attribute has a value that specifies the target of the customization by using an XPath expression.
     - XPath expressions treat XML elements very much like UNIX directory paths. `/element1/element2` and so on would select the all `<element2>` elements inside of `<element1>` elements that are under the root level.
     - `//`: Somewhere underneath the current location (the current location being the root node because nothing comes before it)
     - `[]`: Used to indicate selection criteria
     - `@`: Indicates an attribute

- Examine the remaining customization that specify non-numeric names for the enumerated values of `rankType` and therefore allow xjc to create a RankType Java enum.

Practices for Lesson 4: Exploring SOAP Components

Chapter 4

Practices for Lesson 4: Exploring SOAP Components

Practices Overview

In these practices, you examine a WSDL document in depth and manually create SOAP requests.

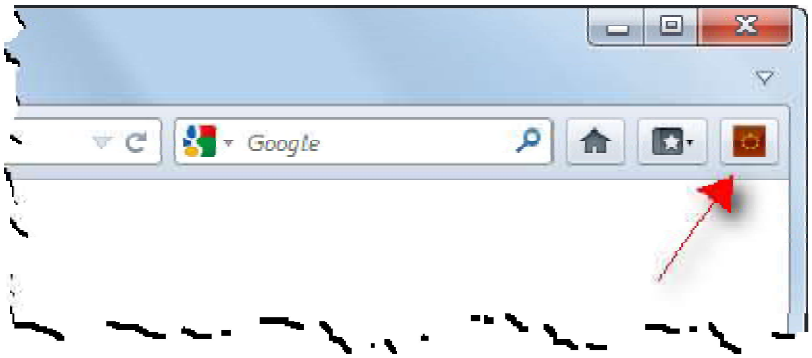
Practice 4-1: Revisiting the Calculator Web Service

Overview

In this practice, you revisit the calculator web service that was briefly explored in Lesson 1.

Tasks

1. Start NetBeans.
2. Open the CalculatorApp project that contains a SOAP web service.
   - Click the File > Open Project menu item.
   - In the Open Project dialog box, select `D:\labs\student\exercises\lab01\CalculatorApp` and click the Open Project button.
3. Right-click the CalculatorApp project and select Deploy.
   - Deploying a Java EE project should automatically start WebLogic Server if it is not already running. WebLogic Server can also be manually stopped and started by using the Services tab in NetBeans.
   - The CalculatorApp provides a simple SOAP-based web service that can be used to add two integers.
4. Launch the RESTClient Firefox extension.
   - Open the Firefox web browser.
   - Start the RESTClient extension by clicking the button in the upper-right corner of Firefox.



5. View the CalculatorApp's WSDL file using the RESTClient extension.
   - Enter http://localhost:7001/CalculatorApp/CalculatorWSService?wsdl into the URL form field and click
   - SEND.
   - View the Response Body (Highlight).
   - The RESTClient extension is used instead of viewing the WSDL in a normal Firefox tab because Firefox will typically hide some information such as namespace declarations like `xmlns:ns="URI"`.
   - In the `<definitions>` find the attribute that sets the default namespace. All non-prefixed elements such as `<definitions>`, `<types>`, and so on belong to this namespace.

     ```
     xmlns="http://schemas.xmlsoap.org/wsdl/"
     ```

   - You should see five element types inside the definitions element.
     - `<types>`: Contains the application-specific XML Schema(s).

- `<message>`: Each SOAP request or response contains an application-specific message. Those messages are declared using a `<message>` element. For each unique request or response that can be processed by the web service, you will see a `<message>` element.
- `<portType>`: The `<portType>` element names the web services available, the operation the web service provides, and which messages are used as input and output for each operation.
- `<binding>`: The `<binding>` element ties the web service to a transport protocol (normally HTTP) and specifies the SOAP body format and encoding.
- `<service>`: The `<service>` element specifies the location (URL) that the web service is available at.

6. Inspect the elements of the WSDL file in more detail and construct a SOAP request.
   - In the `<types>` element, the application-specific schemas will appear. The schemas can be embedded directly in the WSDL or an `<xsd:import>` can be used to reference an external XML Schema.

```
<xsd:schema>
<xsd:import namespace="http://calculator.me.org/"
schemaLocation="http://localhost:7001/CalculatorApp/CalculatorWS
Service?xsd=1"/>
</xsd:schema>
```

   - Open another instance of the RESTClient Firefox extension.
   - Enter the `schemaLocation` value into the RESTClient URL and click SEND. If you view the Response Body (Highlight) you should see:

```
<?xml version='1.0' encoding='UTF-8'?>
<!-- Published by JAX-WS RI at http://jax-ws.dev.java.net. RI's
version is JAX-WS RI 2.2.6hudson-86  svn-revision#12773. -->
<xsd:schema xmlns:ns0="http://calculator.me.org/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://calculator.me.org/">
  <xsd:complexType name="addResponse">
    <xsd:sequence>
      <xsd:element name="return" type="xsd:int"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="add">
    <xsd:sequence>
      <xsd:element name="i" type="xsd:int"/>
      <xsd:element name="j" type="xsd:int"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:element name="addResponse" type="ns0:addResponse"/>
  <xsd:element name="add" type="ns0:add"/>
</xsd:schema>
```

- If you review the schema, you should see two global elements that could be present in an XML document constrained by this schema: `<add>` and `<addResponse>`.
- Begin creating an XML document that will contain a SOAP request.
  - Switch to the Favorites tab in NetBeans.
  - In the `exercises/lab04` folder, create a new empty file named `soaprequest.xml`.
  - Add the standard XML preamble.

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

  - Only two global elements are allowed by the schema, `<add>` and `<addResponse>`. Because `<add>` is used to make the SOAP request, you should create an `<add>` element complete with all required attribute and child elements.
  - Use `calc` as the namespace prefix for the http://calculator.me.org/ namespace.
  - Use a `xsi:schemaLocation` attribute to enable validation of the document in NetBeans.
  - Note that because the schema does not specify `elementFormDefault="qualified"`, only the global elements need to be qualified with a namespace (via prefix or default namespace). Notice that `<i>` and `<j>` have no prefix whereas `<calc:add>` does.

```
<calc:add xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xmlns:calc="http://calculator.me.org/"
          xsi:schemaLocation="http://calculator.me.org/
http://localhost:7001/CalculatorApp/CalculatorWSService?xsd=1">
    <i>7</i>
    <j>10</j>
</calc:add>
```

- Wrap the `<calc:add>` element in a basic SOAP message. A SOAP message contains the same structure for both requests and responses. The root element in a SOAP message is an `<Envelope>`. An `<Envelope>` contains a `<Body>` element and possibly other optional elements such as `<Header>`. The `<Body>` element contains the application-specific XML.

```
<?xml version='1.0' encoding='UTF-8'?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xsi:schemaLocation="http://schemas.xmlsoap.org/soap/envelope/
 http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <!-- calc element goes here -->
    </soap:Body>
</soap:Envelope>
```

- Save your changes.

7. Install the SOAP XML Schema in NetBeans and validate the XML document.
   - Many XML schemas are available for download from the Internet.

- Open the Firefox web browser and go to the URL http://schemas.xmlsoap.org/soap/envelope/
- Save the web page as `D:\labs\student\resources\soap.xsd`.
- In NetBeans, open the Tools > DTDs and XML Schemas menu item.
- Select User Catalog from the list of DTD and XML Schema catalogs.
- Click the Add Local DTD of Schema button.
- Select System ID and enter a value of http://schemas.xmlsoap.org/soap/envelope/.
- For the URI field, click the Browse button and select `D:\labs\student\resources\soap.xsd`.
- Click OK.
- Click Close.
- XML Schema catalogs are used by NetBeans to validate XML documents. By having locally installed XML Schemas, you can validate XML even when NetBeans cannot download schemas on demand, such as in the case of proxy settings preventing access or having no Internet connection. It is a good practice to install any frequently used XML Schemas in a NetBeans catalog.
- Right-click in the `soaprequest.xml` document and select Validate XML.
- The `xsi:schemaLocation` value in `soaprequest.xml` specifies system ID and location pairs for a schema(s). NetBeans searches its catalogs for a schema with a matching system ID and uses the corresponding locally installed schema. If no local schema is found, then the location value from `xsi:schemaLocation` is used to attempt to download the schema. Not all schemas use a matching system ID and location value like the SOAP schema.

8. Submit your SOAP request.
   - View the WSDL file and find the address of the calculator web service (in the `<service>` section).

     ```
     http://localhost:7001/CalculatorApp/CalculatorWSService
     ```
   - Open another instance of the RESTClient Firefox extension.
   - Change the HTTP Method to `POST`.
   - Enter the service location value into the RESTClient URL.
   - Paste your completed SOAP request into the Body field.
   - Click SEND.
   - The response body should be empty. If you view the response headers, you will see a status code of “415 Unsupported Media Type”.
   - Using the Headers menu, add a Custom Header named `Content-Type` with a value of `text/xml`.
   - Click SEND again.
   - The response header should now have a status code of 200 , and if you view the Response Body (Highlight) you should see a SOAP response with the result of the add operation.

     ```
     <?xml version='1.0' encoding='UTF-8'?>
     <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
       <S:Body>
         <ns0:addResponse xmlns:ns0="http://calculator.me.org/">
     ```

```
      <return>17</return>
    </ns0:addResponse>
  </S:Body>
</S:Envelope>
```

Practice 4-2: Configuring WebLogic for WS-* Web Services

Overview

In this practice, you configure your domain for advanced web services features. The WebLogic Web Services stack supports many web services features beyond those required by JAX-WS. Some WS-* extensions require special configuration of WebLogic Server such as:

- Asynchronous messaging
- Web services reliable messaging
- Message buffering
- Security using WS-SecureConversation

A template is available for use during the installation of WebLogic Server, which can automatically configure your server to support these additional features. The same template can also be applied after installation, which is what you do in this practice.

Tasks

1. Stop WebLogic Server.
   - In NetBeans, switch to the Services tab.
   - Expand the Servers branch.
   - Right-click Oracle WebLogic Server and select Stop.
2. Configure your domain for advanced web services features.
   - When creating or extending a domain, you can apply the WebLogic Advanced Web Services for JAX-WS Extension template (`wls_webservice_jaxws.jar`) to automatically configure the resources required to support advanced web services features.

   To apply the same template to an existing server, perform the following steps

   - Open a command prompt (DOS box). Begin by configuring the environment in the command window.

     Note: Copying and pasting to the command prompt is known to cause errors, so manually type the following commands or use the `D:\labs\student\exercises\lab04\wsse.bat` script.

     ```
     D:
     set JAVA_HOME=D:\PROGRA~2\Java\jdk1.7.0_17
     set MW_HOME=D:\weblogic\wls
     set JAVA_VENDOR=Sun
     %MW_HOME%\wlserver\server\bin\setWLSEnv.cmd
     ```

   - Start the WebLogic Scripting Tool (WLST). WLST provides a powerful, scriptable, command-line interface to WebLogic Server.

     ```
     %JAVA_HOME%\bin\java.exe weblogic.WLST -i
     ```

   - Execute the following WLST statements to apply the `wls_webservice_jaxws.jar` template to your existing domain.

     ```
     readDomain('D:/weblogic/domain')
     installDir = 'D:/weblogic/wls/wlserver'
     templateLocation = installDir + \
     '/common/templates/applications/wls_webservice_jaxws.jar'
     ```

```
addTemplate(templateLocation)
updateDomain()
closeDomain()
exit()
```

- Because you are running only a developer install of WLS, there are no other steps to perform. If you are configuring a production server, read http://docs.oracle.com/cd/E24329_01/web.1211/e24964/setenv.htm#CACHCAJD.
- Close the command window.

3. Using the Services tab in NetBeans, start your Oracle WebLogic Server instance. The next practice demonstrates a WSDL file with WS-Policy enhancements and the WS-MakeConnection extension. You must correctly apply the `wls-webservice_jaxws.jar` template in order for the WS-MakeConnection demo to succeed.

Practice 4-3: Exploring SOAP and WSDL Documents with WS-* Extensions

Overview

In this practice, you use a provided project (SlowCalculatorApp) to explore the changes in SOAP and WSDL elements when using WS-* extensions.

WS-* extensions has addition features (and restrictions) for SOAP-based web services. You explore using some of the security-related extensions in a later lesson. For now, you see the WS-Policy statements that are added to a WSDL document when applying a policy to a service.

The WS-* extension used in this example is WS-MakeConnection. WS-MakeConnection enables client polling for web service operations which take a long period of time to execute.

Assumptions

You have completed Practice 4-2 to enable support for advanced web services features in WebLogic.

Tasks

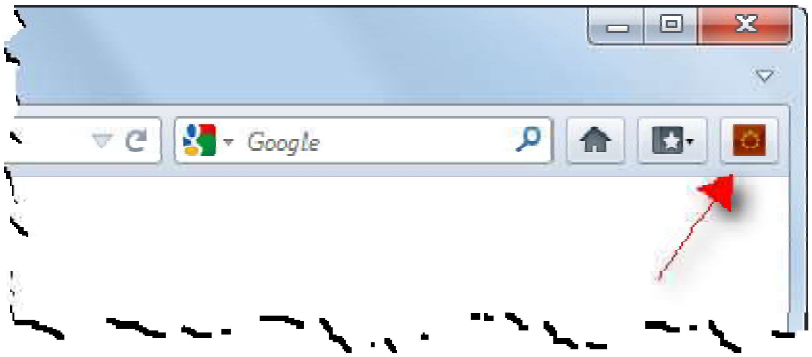
1. You will be supplied a sample project that uses annotation to add a WS-Policy to a web service. Because WS-* extensions are not a mandatory element of JAX-WS and they are primarily geared toward non-function requirements, you will find that the method(s) used by application servers to enable and use these features (if they are supported at all) vary.
2. WebLogic provides two ways to attach a policy to a JAX-WS web services. As an administrator, you can deploy the web service and use the Administration Console to add a policy to a web service. As a developer, you can add WebLogic annotations to web service endpoint classes.
3. To enable the use of the WebLogic WS-Policy annotation, create a new library in NetBeans.
   - Open the Tools menu and choose Ant Libraries.
   - Create a new library.
   - Name: WebLogic Web Services API
   - Library Type: Server Libraries
   - While WebLogic Web Services API is selected, click Add JAR/Folder and add `D:\weblogic\wls\modules\ws.api_2.0.0.0.jar`.
   - Click OK.
4. Open the SlowCalculatorApp project.
   - Click the File > Open Project menu item.
   - In the Open Project dialog box, select `D:\labs\student\exercises\lab04\SlowCalculatorApp` and click the Open Project button.
5. View the `CalculatorWS.java` file, but do not change anything. You should note two things:
   - The `@Policy` annotation at the class level and its import statement. Currently the annotation cannot be found because it is not a part of the standard Java EE libraries.

     Note that `@Policy(uri="policy:Mc1.1.xml", attachToWsdl=true)` specifies that a prewritten policy file included in WebLogic Server should be used (because the

URI begins with `policy:`). To view the prewritten policies, look in your project at Libraries > Oracle WebLogic Server > weblogic.jar > weblogic.wsee.policy.runtime.

- The `CalculatorWS.add` method takes 10 seconds to complete because of a call to Thread.sleep.

6. If it is not already present, add the WebLogic Web Services API library to the SlowCalculatorApp.
   - In the SlowCalculatorApp project, right-click the Libraries branch and choose Add Library.
   - Find and select WebLogic Web Services API and click Add Library.
7. Right-click the SlowCalculatorApp project and select Deploy.
   - Deploying a Java EE project should automatically start WebLogic Server if it is not already running. WebLogic Server can also be manually stopped and started by using the Services tab in NetBeans.
   - The SlowCalculatorApp provides a simple SOAP-based web service that can be used to add two integers.
8. Launch the RESTClient Firefox extension.
   - Open the Firefox web browser.
   - Start RESTClient extension by clicking the button in the upper-right corner of Firefox.



9. View the WS-Policy additions to the SlowCalculatorApp's WSDL file by using the RESTClient extension.
   - Enter http://localhost:7001/SlowCalculatorApp/CalculatorWSService?wsdl into the URL form field and click SEND.
   - View the Response Body (Highlight).
   - You should see the WS-Policy additions to the WSDL file (when compared to the CalculatorApp WSDL from Practice 4-1). An example of one change:

```
<wsp:UsingPolicy wssutil:Required="true"/>
<wsp:Policy wssutil:Id="Mc1.1.xml">
<ns1:MCSupported xmlns:ns1="http://docs.oasis-open.org/ws-rx/wsmc/200702" wsp:Optional="true"/>
</wsp:Policy>
```

10. Submit your SOAP request.
    - View the WSDL file and find the address of the calculator web service (in the `<service>` section).

```
http://localhost:7001/SlowCalculatorApp/CalculatorWSService
```

    - Open another instance of the RESTClient Firefox extension.

- Change the HTTP Method to `POST`.
- Enter the service location value into the RESTClient URL.
- Enter your SOAP request into the Body field. Use the same request as in Practice 4-1 or use the sample below.

```
<env:Envelope
xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
  <env:Header />
  <env:Body>
  <add xmlns="http://calculator.me.org/">
    <i xmlns="">3</i>
    <j xmlns="">9</j>
  </add>
  </env:Body>
</env:Envelope>
```

- Using the Headers menu, add a Custom Header named `Content-Type` with a value of `text/xml`.
- Click SEND.
- After 10 seconds, the response header should have a status code of 200 OK, and if you view the Response Body (Highlight) you should see a SOAP response with the result of the add operation.

```
<?xml version='1.0' encoding='UTF-8'?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <ns0:addResponse xmlns:ns0="http://calculator.me.org/">
      <return>12</return>
    </ns0:addResponse>
  </S:Body>
</S:Envelope>
```

- Because the client is not transmitting any WS-MakeConnection information in its request and the policy makes the use of WS-MakeConnection optional, you are seeing the same response as in Practice 4-1 except it takes 10 seconds to be delivered.

11. Send a WS-MakeConnection request to the SlowCalculatorApp's web service.
    - Using NetBean's Favorites tab, open the `D:\labs\student\exercises\lab04\ws-mc-soapreq.xml` file.
    - The `id` value in the `<wsa:Address>` element specifies a unique client ID that will be used by a client when checking on the progress of their request.
    - Paste the content of `ws-mc-soapreq.xml` into the Body field of the Firefox RESTClient instance from step 10.
    - Click the SEND button to submit the WS-MakeConnection enabled request.
    - The response body you receive should be blank; however, if you view the response headers you should see a status code of 202 Accepted.
    - Click SEND one more time.
12. Send a WS-MakeConnection polling request to the SlowCalculatorApp's web service.

- Using the NetBean's Favorites tab open the `D:\labs\student\exercises\lab04\ws-mc-poll.xml` file.
- The `id` value in the `<wsa:Address>` element (moved from the headers to the body) specifies the same unique client ID that was used to submit the request in step 11.
- Paste the content of `ws-mc-poll.xml` into the Body field of the Firefox RESTClient.
- Click the SEND button to submit the WS-MakeConnection enabled polling request.
- If more than 10 seconds have passed since submitting the requests in step 11, then you should get a 200 OK response with a message body that contains the result of the add operation.
- If additional results are already available, you will see a SOAP header element of `<wsmc:MessagePending>` in the response.
- If no results are available (because not enough time has passed or because no operations have been called), then you will receive a response with no body and a 202 Accepted HTTP status code.

Practices for Lesson 5: Creating JAX-WS Clients

Chapter 5

Practices for Lesson 5: Creating JAX-WS Clients

Practices Overview

In these practices, you set up a basic SOAP web service and create Java clients of that web service.

Practice 5-1: Selecting the JAXB Data Binding and JAXB Providers

Overview

In this practice, you configure the JAXB implementation, which will be used by all applications running in a WebLogic Server instance. WebLogic Server 12*c* is shipped with two JAXB implementations, EclipseLink MOXy and the GlassFish implementation.

MOXy is the default JAXB implementation used by WebLogic Server. The GlassFish reference implementation was the default in previous WebLogic Server releases. In this practice you will reconfigure WebLogic Server to use the GlassFish JAXB reference implementation (RI).

Tasks

1. WebLogic Server 12*c* already includes the GlassFish JAXB RI. You only need to modify two configuration files to switch JAXB implementations. The files are:
   - `META-INF/services/javax.xml.bind.JAXBContext`

     This file specifies the JAXB Provider and determines which implementation is returned when you attempt to call `JAXBContext.newInstance()`.
     - When using MOXy, the value of this text file will be:

     ```
     org.eclipse.persistence.jaxb.JAXBContextFactory
     ```

     - When using the GlassFish RI, the value of this text file will be:

     ```
     com.sun.xml.bind.v2.ContextFactory
     ```

   - `META-INF/services/com.sun.xml.ws.spi.db.BindingContextFactory`

     This file is specifies the JAXB Data Binding Provider and determines which implementation is used by web services.
     - When using MOXy, the value of this text file will be:

     ```
     com.sun.xml.ws.db.toplink.JAXBContextFactory
     ```

     - When using the GlassFish RI, the value of this text file will be:

     ```
     com.sun.xml.ws.db.glassfish.JAXBRIContextFactory
     ```

2. If you wanted to use only the GlassFish RI JAXB implementation, you could add these two files to your application (WAR file). If you did so, you must instruct WebLogic Server to use your version of the files instead of the versions that are included in the server. You do this by adding the files to your application and editing the application's `weblogic.xml` file to include the appropriate `<prefer-application-resources>` block.
3. For this practice, you switch all applications running on WebLogic Server to use the GlassFish JAXB RI.
4. Shut down NetBeans and WebLogic Server. If WebLogic Server was started from within NetBeans, it should shut down automatically when exiting NetBeans. This step is needed because you will be modifying a file that is in use by both NetBeans and WebLogic Server.
5. Using the file explorer, open the `D:\weblogic\wls\wlserver\server\lib` directory and find the `weblogic.jar` file.
6. JAR files are ZIP files can be opened by any tool that supports ZIP files. Inside the `weblogic.jar` file you should see a `META-INF\services\` directory that contains the two files mentioned in task 1.

   Note: For more information about service loaders and the `META-INF\services` directory, see http://docs.oracle.com/javase/7/docs/api/java/util/ServiceLoader.html.

7. Instead of editing the `weblogic.jar` file directly (and potentially breaking your installation), you will back up the existing `weblogic.jar` and install a new one with the changes to the two text files already in place.
8. Rename the `D:\weblogic\wls\wlserver\server\lib\weblogic.jar` file to `weblogic.jar.old`.
9. Copy the `D:\labs\student\exercises\lab05\weblogic.jar` file to the `D:\weblogic\wls\wlserver\server\lib` directory.
10. Start NetBeans.
11. Using the Services tab in NetBeans, start Oracle WebLogic Server. The server should start with no errors.

    Note: A partial test of your changes would be to run the following lines of code within a Java EE module deployed to WebLogic Server.

```
JAXBContext jb = JAXBContext.newInstance("SOME.JAXB.PACKAGE");
System.out.println(jb.getClass().getName());
```

12. For more information see http://docs.oracle.com/cd/E24329_01/web.1211/e24964/data_types.htm#CIHBHDGI.

Practice 5-2: Creating a Card Deck Web Service

Overview

In this practice, you use your existing JAXB annotated playing card classes to create a basic web service out of an EJB.

Tasks

1. Start NetBeans.
2. Create a new Web Application project named CardDecksWS in the `D:\labs\student\exercises\lab05` folder.
   - From the NetBeans menu choose File > New Project.
   - Choose a Category of Java Web.
   - Choose a Project type of Web Application.
   - Click Next.
   - Change the Project Name to CardDecksWS.
   - Change the Project Location to `D:\labs\student\exercises\lab05`.
   - Click Next.
   - Deselect Set Source Level to 6.
   - Click Finish.
3. Change the Source/Binary Format to JDK7 for the CardDecksWS project.
   - WebLogic Service 12*c* is certified to run on JDK7 and you will use Java 7 features.
   - Right-click the CardDecksWS project and choose Properties.
   - In the Sources category, change the Source/Binary Format to JDK7.
   - Click OK.
4. Copy the playingcards package from the JavaToSchema project to the CardDecksWS project.
   - Using NetBeans, open the JavaToSchema project from Practice 3-2.
   - Copy the playingcards package from the JavaToSchema project to the CardDecksWS project.
5. Modify the `Card` class to have a no-arg constructor.
   - In the CardDecksWS project, open the `playingcards/Card.java` file.
   - These classes are annotated as JAXB bound POJOs. The `Card` class lacks a no-arg constructor because it was used only to marshall Java to XML.
   - Add the required no-arg constructor that allows JAXB to unmarshall a Card.

```
public Card() { }
```

6. Create the `ejbs.CardDeckSessionBean` singleton EJB class.
   - Session EJBs often function as a “service” in an application. Here you will create an EJB and see how it can be easily turned into a web service.
   - Right-click the Source Packages node of the CardDecksWS project and choose New > Other.
   - Select a Category of Enterprise JavaBeans and a File Type of Session Bean.
   - Click Next.
   - Change the EJB Name to CardDeckSessionBean.

- Change the package to ejbs.
- Change the Session Type to Singleton.
- Click Finish.

7. The CardDeckSessionBean creates and maintains a collection of card decks.
   - Add an `int` field to keep track of card deck IDs and a `HashMap` field to store decks using their unique ID.

```
private int nextDeckId = 0;
private Map<Integer, FrenchCardDeck> decks = new HashMap<>();
```

   - Add a method called `createDeck` which accepts one parameter (the number of jokers) and returns the unique ID of the deck that was created.

```
public Integer createDeck(int jokerCount) {
    int deckId = nextDeckId++;
    decks.put(deckId, new FrenchCardDeck(jokerCount));
    return deckId;
}
```

   - Create a `listDeckIds` method that returns an array of all the deck IDs.

```
public Integer[] listDeckIds() {
    return decks.keySet().toArray(new Integer[0]);
}
```

   - Create a `deleteDeck` method that deletes a deck of cards by ID and returns `true` if the deck was present and deleted.

```
public boolean deleteDeck(int id) {
    FrenchCardDeck deck = decks.remove(id);
    if(deck == null) {
        return false;
    } else {
        return true;
    }
}
```

   - Create a `shuffleDeck` method that shuffles a deck of cards by ID and returns `true` if the deck was present and shuffled.

```
public boolean shuffleDeck(int id) {
    FrenchCardDeck deck = decks.get(id);
    if(deck == null) {
        return false;
    } else {
        deck.shuffle();
        return true;
    }
}
```

   - Create a `getDeck` method that returns a deck (CardCollection) by ID.

```
public CardCollection getDeck(int id) {
    FrenchCardDeck deck = decks.get(id);
    return deck;
}
```

- Add any required import statements.

8. Turn the CardDeckSessionBean into a web service.
   - Add the `@WebService` annotation to the `CardDeckSessionBean` class.

```
@WebService
@Singleton
public class CardDeckSessionBean
```

   - Add any required import statements.
9. Deploy the CardDecksWS project.
10. Test the CardDeckSessionBean web service.
    - After deployment, the web service should be located at http://localhost:7001/CardDeckSessionBean/CardDeckSessionBeanService.
    - The WSDL file can be located at http://localhost:7001/CardDeckSessionBean/CardDeckSessionBeanService?wsdl.
    - Open a web browser and use the WebLogic Server SOAP Test Client to test the CardDeckSessionBean web service. Go to the URL http://localhost:7001/wls_utc/begin.do?wsdlUrl=http://localhost:7001/CardDeckSessionBean/CardDeckSessionBeanService?wsdl.
    - Start by creating card decks with varying amounts of jokers; then test the other operations.

Practice 5-3: Creating a Java SE Web Service Client

Overview

In this practice, you create a Java SE application that uses JAX-WS to call a web service.

Assumptions

The CardDecksWS project from Practice 5-2 is deployed.

Tasks

1. Create a new Java SE Application project named CardDeckSEClient in the `D:\labs\student\exercises\lab05` folder.
   - From the NetBeans menu choose File > New Project.
   - Choose a Category of Java.
   - Choose a Project type of Java Application.
   - Click Next.
   - Change the Project Name to CardDeckSEClient.
   - Change the Project Location to `D:\labs\student\exercises\lab05`.
   - Click Finish.
2. Create a web service client.
   - Right-click the CardDeckSEClient project and choose New > Other.
   - Select a category of Web Services and a file type of Web Service Client.
   - Click Next.
   - Select WSDL URL and enter http://localhost:7001/CardDeckSessionBean/CardDeckSessionBeanService?wsdl.
   - Enter a package name of “cards”.
   - Click Finish.
3. Examine the generated source code.
   - You should see a new Generated Sources (jax-ws) node in the project.
   - Open the cards package in the Generated Sources (jax-ws) node.
   - If you review the source code from the generated classes, you find that most of them are just JAXB annotated classes.
4. Build the CardDeckSEClient project.
   - Right-click the CardDeckSEClient project and choose Build.
   - NetBeans does not always recompile generated sources. If you find that when you attempt to run your application it complains about missing classes, you should make sure the project has been built.
5. Call the CardDeckSessionBean web service.
   - Open the `CardDeckSEClient.java` file and modify the `main` method.
   - Create a reference to the generated CardDeckSessionBeanService.

     ```
     CardDeckSessionBeanService service = new
     CardDeckSessionBeanService();
     ```

   - Use the service to obtain the port.

     ```
     CardDeckSessionBean port = service.getCardDeckSessionBeanPort();
     ```

- Use the port to call the `createDeck` operation to make a card deck with zero jokers.

```
int deckId1 = port.createDeck(0);
```

- Shuffle the deck that was just created.

```
port.shuffleDeck(deckId1);
```

- Retrieve the deck and display its cards.

```
StackType deck = port.getDeck(deckId1);
List<CardType> cards = deck.getCard();

for(CardType card : cards) {
    if(card.getRank().equalsIgnoreCase("JOKER")) {
        System.out.println(card.getColor() + " " +
            card.getRank());
    } else {
        System.out.println(card.getRank() + " of " +
            card.getSuit());
    }
```

6. Generate web service client convenience methods.
   - NetBeans can generate methods for you that hide the details of having to obtain a service and port.
   - Right-click in the source window of CardDeckSEClient and choose Insert Code.
   - Select Call Web Service Operation.
   - Select the `createDeck` operation from CardDeckSessionBeanPort.
   - Click OK.
   - Repeat the process for the `shuffleDeck` and `getDeck` operations.
   - Use the NetBeans generated web service client methods.
   - Comment out the code that directly used the web service and port.

```
//CardDeckSessionBeanService service =
//    new CardDeckSessionBeanService();
//CardDeckSessionBean port =
//    service.getCardDeckSessionBeanPort();
//int deckId1 = port.createDeck(0);
//port.shuffleDeck(deckId1);
//StackType deck = port.getDeck(deckId1);
```

   - Below the commented code, use the NetBeans generated method to perform the actions that are equivalent to the commented out code.

```
int deckId1 = createDeck(0);
shuffleDeck(deckId1);
StackType deck = getDeck(deckId1);
```

7. Run the project.

Practice 5-4: Creating a Java EE Web Service Client

Overview

In this practice, you create a Java EE application that uses a JAX-WS proxy to call a web service.

Assumptions

The CardDecksWS project from Practice 5-2 is deployed.

Tasks

1. Create a new web application project named CardDeckEEClient in the `D:\labs\student\exercises\lab05` folder.
   - From the NetBeans menu choose File > New Project.
   - Choose a Category of Java Web.
   - Choose a Project type of Web Application.
   - Click Next.
   - Change the Project Name to CardDeckEEClient.
   - Change the Project Location to `D:\labs\student\exercises\lab05`.
   - Click Next.
   - Deselect Set Source Level to 6.
   - Click Finish.
2. Change the Source/Binary Format to JDK7 for the CardDeckEEClient project.
   - WebLogic Service 12*c* is certified to run on JDK7 and you will use Java 7 features.
   - Right-click the CardDeckEEClient project and choose Properties.
   - In the Sources category, change the Source/Binary Format to JDK7.
   - Click OK.
3. Create a web service client.
   - Right-click the CardDeckEEClient project and choose New > Other.
   - Select a category of Web Services and a File Type of Web Service Client.
   - Click Next.
   - Select WSDL URL and enter http://localhost:7001/CardDeckSessionBean/CardDeckSessionBeanService?wsdl.
   - Enter a package name of “cards”.
   - Click Finish.
4. Examine the generated source code.
   - You should see a new Generated Sources (jax-ws) node in the project.
   - Open the cards package in the Generated Sources (jax-ws) node.
   - If you review the source code from the generated classes, you find that most of them are just JAXB annotated classes.
5. Create the web.WSClient servlet.
   - Right-click the CardDeckEEClient project and choose New > Other.
   - Select a category of Web and a File Type of Servlet.
   - Click Next.

- Enter a class name of WSClient and a package name of web.
- Click Finish.

6. Call the CardDeckSessionBeanService from the WSClient servlet.
   - Add a `@WebServiceRef` annotated field to the `WSClient` class.

```
@WebServiceRef(wsdlLocation = "WEB-
INF/wsdl/localhost_7001/CardDeckSessionBean/CardDeckSessionBeanS
ervice.wsdl")
private CardDeckSessionBeanService service;
```

   - When you added the web service reference to the project in step 2, a local copy of the WSDL was added. Using the local copy for the `wsdlLocation` eliminates a network request and improves performance.
   - At the beginning of the `processRequest` method, use the web service reference you just created to create a card deck with two jokers, shuffle it, and retrieve the shuffled cards. Remember the injected service is not thread-safe.

```
StackType deck;
synchronized(this) {
    CardDeckSessionBean port =
service.getCardDeckSessionBeanPort();
    int deckId1 = port.createDeck(2);
    port.shuffleDeck(deckId1);
    deck = port.getDeck(deckId1);
}
```

   - Display the card ranks in the body of the HTML output.

```
for(CardType card : deck.getCard()) {
    out.println("Card: " + card.getRank());
}
```

7. Build the CardDeckEEClient project.
   - Right-click the CardDeckEEClient project and choose Build.
   - NetBeans does not always recompile generated sources. If you find that when you attempt to run your application it complains about missing classes, you should make sure the project has been built.
8. Run the WSClient Servlet.
   - Right-click the WSClient servlet and choose Run File.
   - Click OK in the Set Servlet Execution URI dialog box.
9. Use the playing card Unicode block and DejaVuSans font to display the playing cards.
   - Using the Favorites tab, copy the `DejaVuSans.ttf` font to the CardDeckEEClient Web Pages.
     - On the Favorites tab, expand exercises > lab05 > dejavu-fonts-ttf-2.33.zip > dejavu-fonts-ttf-2.33.ttf.
     - Right-click `DejaVuSans.ttf` and choose Copy.
     - Switch back to the Project tab and paste the font to the Web Pages folder of your project.
   - Create the `card.css` file.

- Create a new Cascading Style Sheet file named card. The `.css` extension is automatically added when using the Cascading Style Sheet file type in the Web category.
- Modify the `card.css` file to create CSS classes that use the DejaVusSans font.

```
@font-face {
    font-family: 'DejaVu Sans';
    src: url('DejaVuSans.ttf');
}
.card {
    font-family: 'DejaVu Sans';
    font-size: 150%;
}
.redcard {
    font-family: 'DejaVu Sans';
    font-size: 150%;
    color: red;
}
```

- Copy the `CardUtils.java` file to your project.
  - Create the `util` package.
  - Switch to the Favorites tab and copy the `D:\labs\student\exercises\lab05\CardUtils.java` file.
  - Switch to the Projects tab and paste the `CardUtils.java` file into the util package.
  - The `Card` class in the CardDeckEEClient project (most likely called `CardType`) was generated by wsimport based solely on the WSDL file and any referenced XML Schemas. The client-side `CardType` class is lacking any business logic that was in the `Card` class that is internal to the web service.
  - The `CardUtils` class provides methods to determine the color of a card (only Jokers are transmitted across the wire with a color) and convert a card to its Unicode value in the playing card block.

    Note: If your generated JAX-WS client code is different than what is expected by the CardUtils source, you may have to adjust the CardUtils code.
- Modify the WSClient servlet to use the CSS classes.
  - In the HTML header, print out the stylesheet link.

```
out.println("<link rel='stylesheet' type='text/css'
href='card.css'>");
```

  - Replace the existing `for` loop with a new one that uses the `CardUtils` class and CSS classes to print out each card as a Unicode symbol.

```
for(CardType card : deck.getCard()) {
    if(CardUtils.getColor(card) == ColorType.BLACK) {
        out.println("<span class='card'>" +
CardUtils.toUnicode(card) + "</span>");
    } else {
```

```
        out.println("<span class='redcard'>" +
CardUtils.toUnicode(card) + "</span>");
    }
}
```

10. Run the WSClient Servlet.
    - Right-click the WSClient servlet and choose Run File.
    - Click OK in the Set Servlet Execution URI dialog box.

(Optional) Practice 5-5: Binding Customization

Overview

In this practice, you customize the binding to improve the method and class names of the generated JAX-WS artifacts. A lot of the work is just JAXB binding customization just like you already performed in Lesson 3.

Assumptions

The CardDecksWS project from Practice 5-2 is deployed and you completed the CardDeckSEClient project from Practice 5-3.

Tasks

1. Open the local WSDL and XSD files in the CardDeckSEClient project.
   - When you add a Web Service Client to a project, the WSDL and XSD files for the remote web service are copied to your project. The files are copied to two locations, only the files in the xml-resources folder are used when running wsimport (build project).
   - Switch to the Files tab in NetBeans.
   - Expand the folder in the CardDeckSEClient project starting with the xml-resources folder until you find the WSDL and XSD files.
   - You can modify these files and/or add external binding files.

   Warnings:
   - If you modify the local WSDL or XSD files, you should not make them incompatible with the remote web service. In other words, add only in-line binding customizations.
   - If you modify the local WSDL or XSD files, you could overwrite all your customizations when you refresh the web service reference. When you right-click Web Service Reference > CardDeckSessionBeanService and choose Refresh to regenerate the JAX-WS client code, be sure that the check box to replace your local copy is NOT selected.
2. Customize the `CardDeckSessionBeanService.xsd_1.xsd` file with JAXB inline binding customizations.
   - Open the `CardDeckSessionBeanService.xsd_1.xsd` file (under xml-resources).
   - Add the required attributes to the `xs:schema` element.

     ```
     xmlns:jaxb="http://java.sun.com/xml/ns/jaxb" jaxb:version="2.1"
     ```
   - Modify the `stackType` complex type to have a class name of `CardStack` (instead of `StackType`) and to have a `getCards()` method instead of `getCard()`.

     ```
     <xs:complexType name="stackType">
         <xs:annotation>
             <xs:appinfo>
                 <jaxb:class name="CardStack"/>
             </xs:appinfo>
         </xs:annotation>
         <xs:sequence>
             <xs:element name="card" type="tns:cardType"
     maxOccurs="unbounded">
                 <xs:annotation>
     ```

```
                <xs:appinfo>
                    <jaxb:property name="cards"/>
                </xs:appinfo>
            </xs:annotation>
        </xs:element>
    </xs:sequence>
</xs:complexType>
```

3. Execute wsimport.
   - Switch back to the Projects tab.
   - Expand Web Service References for your project.
   - Right-click CardDeckSessionBeanService and choose Refresh. Make sure that the option to replace your local WSDL file with the original is NOT selected.
4. Modify the CardDeckSEClient to use the new class and method names.
5. Build your project.
6. Run the project.

(Optional) Practice 5-6: Creating a JAX-WS Dispatch Web Service Client

Overview

In this practice, you create a Java SE application that uses a JAX-WS dispatch client to call a web service. Note that no generated artifacts are used in this project. Though the code does make some assumptions about the structure of the request and response messages, you could also programmatically parse the WSDL and associated XML Schemas to create a truly dynamic client.

Assumptions

The CardDecksWS project from Practice 5-2 is deployed.

Tasks

1. Create a new Web Application project named CardDeckSEDispatchClient in the `D:\labs\student\exercises\lab05` folder.
   - From the NetBeans menu choose File > New Project.
   - Choose a Category of Java.
   - Choose a Project type of Java Application.
   - Click Next.
   - Change the Project Name to CardDeckSEDispatchClient.
   - Change the Project Location to `D:\labs\student\exercises\lab05`.
   - Click Finish.
2. Call the CardDeckSessionBean web service.
   - Open the `CardDeckSEDispatchClient.java` file and modify the `main` method.
   - Create a QName for the service.
   - Create a QName for the port.
   - Create a String for the service endpoint.

```
QName serviceQName = new QName("http://ejbs/",
"CardDeckSessionBeanService");
QName portQName = new QName("http://ejbs/", "EchoServicePort");
String endpointUrl =
"http://localhost:7001/CardDeckSessionBean/CardDeckSessionBeanSe
rvice";
```

   - Create a Service, add the port to the Service, and create a SOAP 1.1 message.

```
Service cardService = Service.create(serviceQName);
cardService.addPort(portQName, SOAPBinding.SOAP11HTTP_BINDING,
endpointUrl);
Dispatch<SOAPMessage> dispatch =
cardService.createDispatch(portQName, SOAPMessage.class,
Service.Mode.MESSAGE);
MessageFactory mf =
MessageFactory.newInstance(SOAPConstants.SOAP_1_1_PROTOCOL);
SOAPMessage request = mf.createMessage();
SOAPPart part = request.getSOAPPart();
```

```
SOAPEnvelope envelope = part.getEnvelope();
SOAPBody body = envelope.getBody();
```

- Complete the SOAP body with a request to create a deck with one joker.

```
SOAPElement operation = body.addChildElement("createDeck",
"ns1", "http://ejbs/");
SOAPElement value = operation.addChildElement("arg0");
value.addTextNode("1");
SOAPMessage response = dispatch.invoke(request);
String deckId = response.getSOAPBody().getTextContent();
System.out.println("Deck ID: " + deckId);
```

- Create a new SOAP message.

```
request = mf.createMessage();
part = request.getSOAPPart();
envelope = part.getEnvelope();
body = envelope.getBody();
```

- Complete the SOAP body with a request to get the deck previously created.

```
operation = body.addChildElement("getDeck", "ns1",
"http://ejbs/");
value = operation.addChildElement("arg0");
value.addTextNode(deckId);
response = dispatch.invoke(request);
SOAPBody responseBody = response.getSOAPBody();
```

- Loop through the content of the SOAP response body and print out the value of each card.

```
NodeList nodes = responseBody.getChildNodes();
NodeList cardNodes =
nodes.item(0).getChildNodes().item(0).getChildNodes();
for(int i = 0; i < cardNodes.getLength(); i++) {
    Node card = cardNodes.item(i);
    NodeList cardChildNodes = card.getChildNodes();
    String cardStr = "";
    for(int x = 0; x < cardChildNodes.getLength(); x++) {
        Node cardChild = cardChildNodes.item(x);
        cardStr += cardChild.getTextContent() + " ";
    }
    System.out.println(cardStr.trim());
}
```

3. Add a `throws SOAPException` clause to the `main` method.
4. Run your project. You should receive output similar to:

```
Deck ID: 164
A CLUBS
2 CLUBS
```

```
3 CLUBS
4 CLUBS
5 CLUBS
```

(Optional) Practice 5-7: Using WS-MakeConnection with a JAX-WS Client

Overview

In this practice, you create a JAX-WS client which uses a WS-* extension (WS-MakeConnection).

Assumptions

You completed Practice 4-3 (SlowCalculatorApp).

Tasks

1. Open the SlowCalculatorApp project.
   - Click the File > Open Project menu item.
   - In the Open Project dialog box, select `D:\labs\student\exercises\lab04\SlowCalculatorApp` and click the Open Project button.
2. Right-click the SlowCalculatorApp project and select Deploy.
3. Create a new Web Application project named SlowCalculatorEEClient in the `D:\labs\student\exercises\lab05` folder.
   - From the NetBeans menu choose File > New Project.
   - Choose a Category of Java Web.
   - Choose a Project type of Web Application.
   - Click Next.
   - Change the Project Name to SlowCalculatorEEClient.
   - Change the Project Location to `D:\labs\student\exercises\lab05`.
   - Click Next.
   - Deselect Set Source Level to 6.
   - Click Finish.
4. Set the Source/Binary Format to JDK 7.
   - Right-click the project and click Properties.
   - In the Project Properties dialog box, select Sources.
   - Change the Source/Binary Format to JDK 7.
   - ClickOK.
5. Use the WebLogic clientgen Ant task instead of wsimport.
   - Previously, you generated JAX-WS artifacts by adding a Web Service Client by using NetBeans. That activity configures and runs the wsimport command. You will be using an alternative method to generate JAX-WS artifacts.
   - DO **NOT** ADD A WEB SERVICE CLIENT TO THIS PROJECT!
   - Switch to the Files tab.
   - Open the `build.xml` file in the SlowCalculatorEEClient.
   - At the bottom of the `build.xml` but before the closing `</project>` tag, add: (there is a copy/paste example located at `D:\labs\sudent\resources\clientgen_template.xml`)

```
<target name="-pre-compile">
    <echo message="RUNNING clientgen" />
    <taskdef name="clientgen"
classname="weblogic.wsee.tools.anttasks.ClientGenTask">
        <classpath>
            <path
path="${javac.classpath}:${j2ee.platform.classpath}"/>
        </classpath>
    </taskdef>
    <clientgen

wsdl="http://localhost:7001/SlowCalculatorApp/CalculatorWSServic
e?wsdl"
        destDir="${src.dir}"
        packageName="calc"
        copyWsdl="true"
        type="JAXWS">
        <binding file="${src.dir}/jaxws-binding.xml" />
    </clientgen>
    <delete>
        <fileset dir="${src.dir}" includes="**/*.class"/>
    </delete>
</target>
```

- Note the `wsdl` and `packageName` attributes in the `clientgen` element; there are the two pieces of unique information that you must supply for each web service that will be used.

6. Add a binding customization file.
   - The clientgen task you added referenced an optional binding file.
   - Create a new well-formed XML named jaxws-binding in the Source Packages folder.
   - Use the binding file to enable async methods for the web service operations. (There is a copy/paste example located at `D:\labs\sudent\resources\jaxws-binding.xml`)

```
<?xml version="1.0" encoding="UTF-8"?>
<bindings
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"

wsdlLocation="http://localhost:7001/SlowCalculatorApp/Calculator
WSService?wsdl"
    xmlns="http://java.sun.com/xml/ns/jaxws">
    <bindings node="wsdl:definitions">
        <package name="calc"/>
        <enableAsyncMapping>true</enableAsyncMapping>
    </bindings>
```

```
</bindings>
```

7. Inspect the generated async methods.
   - Clean and build the project.
   - Open the `calc.CalculatorWS.java` file and view its source. You should see three `add*(int,int)` methods. Two are asynchronous methods and one is synchronous.
8. Create the `web.CalcServlet` class.
9. Use the calculator web service in the `CalcServlet` class.
   - In the beginning of the `processRequest` method, add:

```
URL wsdlUrl =
new URL("file://./META-INF/wsdls/CalculatorWSService.wsdl");
CalculatorWSService service =
new CalculatorWSService(wsdlUrl);
McFeature mcFeature = new McFeature();
mcFeature.setUseMcWithSyncInvoke(true);
CalculatorWS port = service.getCalculatorWSPort(mcFeature);
```

   - Add any needed imports. If you look at the added import statements you should see that `McFeature` is a weblogic-specific class. Many WS-* extensions (if they are even supported) require the use of vendor-specific classes.
   - Calling `mcFeature.setUseMcWithSyncInvoke(true)` means that WS-MakeConnection will be used even when calling the non-async `add` method.
   - Add a call to `port.add()` and display the result in the body of the HTML output.

```
out.println(port.add(5, 10));
```

10. Run the `CalcServlet` class.
    - The servlet output should take a long time (approximately 10 seconds) to load in your web browser. This is because the web service responds slowly.
    - At this point, you cannot tell if your client is maintaining a connection to the web service or using WS-MakeConnection.
11. Enable HTTP traffic dumping for JAX-WS endpoints on the server.
    - Open the Tools > Servers menu in NetBeans.
    - Select Oracle WebLogic Server.
    - Open the Platform tab.
    - Add a VM Options value of:

```
-Dcom.sun.xml.ws.transport.http.HttpAdapter.dump=true
```

    - Click Close.
12. Restart WebLogic Server.
    - Open the Services tab in NetBeans.
    - Expand Servers.
    - Right-click Oracle WebLogic Server and choose Restart.
    - Wait for the server to finish loading.
13. View the Oracle WebLogic Server output while running the CalcServlet web service client.
    - In the NetBeans Output window, select the Oracle WebLogic Server tab. In the output area of for the tab, right-click and select Clear.

- Run the `CalcServlet` class.
- Quickly switch back to NetBeans and select the Oracle WebLogic Server tab in the Output area.
- You should see the polling of the WS-MakeConnection client in action.

Practices for Lesson 6: Exploring REST Services

Chapter 6

Practices for Lesson 6: Exploring REST Services

Practices Overview

In these practices, you will explore a RESTful web service provided by WebLogic Server.

Practice 6-1: Enabling RESTful Management Services for WebLogic

Overview

In this practice, you enable a REST web service within WebLogic Server that provides advanced monitoring capabilities. Because monitoring, by its very nature, does not change anything, you will find that the REST service explored here uses only `GET` operations. Later lessons will leverage additional HTTP methods.

WebLogic Server is not the only Java EE application server that supports a REST interface. For information on REST administration in GlassFish see http://docs.oracle.com/cd/E26576_01/doc.312/e24928/general-administration.htm#gjipx.

Tasks

1. Start NetBeans if needed.
2. Using the Services tab, start WebLogic Service if it is not already started.
3. View the WebLogic Server Administration Console.
   - You can either right-click Oracle WebLogic Service in the Services tab and choose View Admin Console, or you can directly launch a web browser with a URL on http://localhost:7001/console/.
4. Log in to the Administration Console.
   - Username: `weblogic`
   - Password: `welcome1`
5. In the Domain Structure box on the left side, click mydomain.
6. Make sure the Configuration > General tabs are selected.
7. Expand the Advanced section at the bottom of the page.
8. Check the Enable RESTful Management Services box.
9. Click the Save button.
10. Close the Administration Console.
11. Using the NetBeans Service tab, restart Oracle WebLogic Server.

For more information on enabling RESTful Management Services see http://docs.oracle.com/cd/E24329_01/apirefs.1211/e24401/taskhelp/domainconfig/EnableRESTfulManagementServices.html.

Practice 6-2: Exploring WebLogic RESTful Management Services

Overview

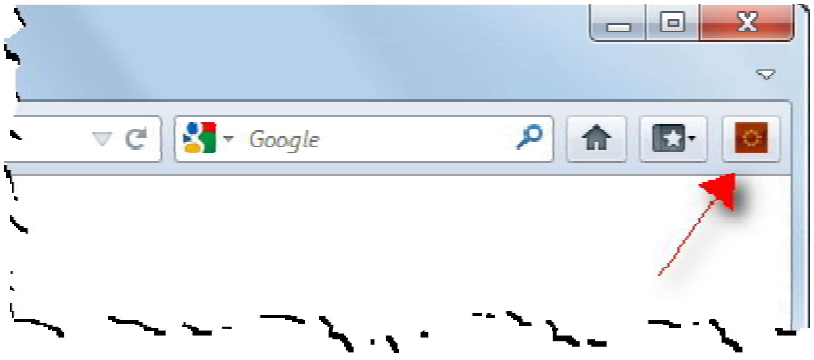
In this practice, you explore the RESTful Management Services provided by WebLogic Server using cURL and the RESTClient extension for Firefox.

Assumptions

You have completed Practice 6-1 to enable RESTful Management Services. You have completed Practice 1-4 and installed the RESTClient Firefox extension.

Tasks

1. WebLogic's RESTful Management Services provide several resources whose representations can be retrieved. All resources URLs begin with: http://localhost:7001/management/tenant-monitoring/.
2. The following resources are available under the URL mentioned in step 1.
   - servers – The status of all servers in a domain
   - servers/{name} – A specific server's status
   - clusters – The status of all clusters in a domain
   - clusters/{name} – A specific cluster's status
   - applications – The status of all applications deployed in a domain
   - applications/{name} – A specific application's status
   - datasources – The status of all data sources in a domain
   - datasources/{name} – The status of a specific data source
3. Using a web browser, open the servers resource by using the URL http://localhost:7001/management/tenant-monitoring/servers/.
4. Each resource is available in three different representation formats.
   - application/xml
   - application/json
   - text/html
5. Start the Firefox web browser.
6. Start the RESTClient extension by clicking the button in the upper-right corner of Firefox.



7. Get the servers resource by using the RESTClient extension.
   - Enter http://localhost:7001/management/tenant-monitoring/servers/ into the URL form field and click SEND.
   - If prompted, enter a username of `weblogic` and a password of `welcome1`.

- View the Response Body (Highlight). Notice the format is text/html.
- Use the Headers menu to add a Custom Header with the values:
  - Name: Accept
  - Value: application/xml
- Click the SEND button again to re-execute the web service. The response should now be an XML response.

8. Using the RESTClient extension, attempt to determine the amount of heap memory in use by each server.

Practice 6-3: Updating Jersey (JAX-RS)

Overview

In this practice, you install Jersey 1.17 as a WebLogic Server shared library. Jersey is the reference implementation of JAX-RS. A shared library is a collection of classes and resources that any deployed application can use. These shared libraries can even take preference over classes that are bundled with WebLogic Server, thereby allowing deployed applications to use different versions of a library. WebLogic Server 12.1.1.1 includes Jersey 1.9 as the JAX-RS implementation. By deploying Jersey 1.17 as a shared library, you enable any deployed application to select the version of Jersey it will use at run time.

A list of changes can be found at http://java.net/projects/jersey/sources/svn/content/tags/jersey-1.17/jersey/changes.txt.

Tasks

1. Start NetBeans if needed.
2. Create a NetBean Ant library for Jersey 1.17.
   - This NetBean Ant library can be used by any application that directly includes Jersey library JAR files such as a Java SE RESTful client application. Java EE applications can take advantage of libraries available in the application server.
   - Select Tools > Ant Libraries from the NetBeans menu.
   - Click the New Library button.
   - Enter the values:
     - Library Name: `Jersey 1.17`
     - Library Type: `Class Libraries`
   - Click OK.
   - Add the following JARs to the Classpath tab of the Jersey 1.17 library.
     - `D:\labs\student\resources\jersey\activation-1.1.1.jar`
     - `D:\labs\student\resources\jersey\asm-3.1.jar`
     - `D:\labs\student\resources\jersey\jackson-core-asl-1.9.2.jar`
     - `D:\labs\student\resources\jersey\jackson-jaxrs-1.9.2.jar`
     - `D:\labs\student\resources\jersey\jackson-mapper-asl-1.9.2.jar`
     - `D:\labs\student\resources\jersey\jackson-xc-1.9.2.jar`
     - `D:\labs\student\resources\jersey\jersey-bundle-1.17.jar`
     - `D:\labs\student\resources\jersey\jettison-1.1.jar`
     - `D:\labs\student\resources\jersey\jsr311-api-1.1.1.jar`
     - `D:\labs\student\resources\jersey\oauth-client-1.17.jar`
     - `D:\labs\student\resources\jersey\oauth-server-1.17.jar`
     - `D:\labs\student\resources\jersey\oauth-signature-1.17.jar`
   - Switch to the Javadoc tab of the Jersey 1.17 library and add the folders:
     - `D:\labs\student\resources\jersey\apidocs\jersey`
     - `D:\labs\student\resources\jersey\apidocs\jsr311`
     - `D:\labs\student\resources\jersey\apidocs\contribs\oauth\signature`

- `D:\labs\student\resources\jersey\apidocs\contribs\oauth\server`
- `D:\labs\student\resources\jersey\apidocs\contribs\oauth\client`

- Click OK.

3. Using the Services tab, start WebLogic Service if it is not already started.
4. View the WebLogic Server Administration Console.
   - You can either right-click Oracle WebLogic Service in the Services tab and choose View Admin Console, or you can directly launch a web browser with a URL on http://localhost:7001/console/.
5. Log in to the Administration Console.
   - Username: `weblogic`
   - Password: `welcome1`
6. Install `Jersey117_SharedLibrary.war` as a shared library.
   - On the left hand side, click on Deployments (Domain Structure > mydomain > Deployments)
   - Click the Install button.
   - Specify a Path of `D:\labs\student\projects\Jersey117_SharedLibrary\dist\Jersey117_SharedLibrary.war`
   - Click Next.
   - Choose “Install this deployment as a library”.
   - Click Next.
   - Click Finish.
   - On the Deployments page, you should see Jersey-1.17(1.17,1.17.0) listed with a Type of Library.
7. Using this shared library in any Java EE web application requires the following lines to be added to the `weblogic.xml` descriptor.

```
<library-ref>
  <library-name>Jersey-1.17</library-name>
  <specification-version>1.17</specification-version>
  <implementation-version>
1.17.0</implementation-version>
  <exact-match>true</exact-match>
</library-ref>
```

   - For more information see http://docs.oracle.com/cd/E24329_01/web.1211/e24983/version.htm.

Practice 6-4: Creating a Basic RESTful Web Service with JAX-RS

Overview

In this practice, you create a simple RESTful web service with JAX-RS. A later lesson will cover JAX-RS in-depth.

Tasks

1. Start NetBeans.
2. Create a new web application project named HelloRest in the `D:\labs\student\exercises\lab06` folder.
   - From the NetBeans menu choose File > New Project.
   - Choose a Category of Java Web.
   - Choose a Project type of Web Application.
   - Click Next
   - Change the Project Name to HelloRest.
   - Change the Project Location to `D:\labs\student\exercises\lab06`.
   - Click Next
   - Deselect Set Source Level to 6.
   - Click Finish.
3. Change the Source/Binary Format to JDK7 for the HelloRest project.
   - WebLogic Service 12*c* is certified to run on JDK7 and you will use Java 7 features.
   - Right-click the HelloRest project and choose Properties.
   - In the Sources category, change the Source/Binary Format to JDK7.
   - Click OK.
4. Add the shared library reference for Jersey 1.17.
   - In the `weblogic.xml` configuration file, add a `library-ref` element that instructs WebLogic Server to use the previously installed shared library when running this application.
   - Open the `weblogic.xml` file (under Configuration Files) and add the library reference.

```
<?xml version="1.0" encoding="UTF-8"?>
<weblogic-web-app
xmlns="http://xmlns.oracle.com/weblogic/weblogic-web-app"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd
http://xmlns.oracle.com/weblogic/weblogic-web-app
http://xmlns.oracle.com/weblogic/weblogic-web-app/1.0/weblogic-
web-app.xsd">
    <jsp-descriptor>
        <keepgenerated>true</keepgenerated>
        <debug>true</debug>
    </jsp-descriptor>
    <context-root>/HelloRest</context-root>
```

```
    <library-ref>
        <library-name>Jersey-1.17</library-name>
        <specification-version>1.17</specification-version>
        <implementation-version>1.17.0</implementation-version>
        <exact-match>true</exact-match>
    </library-ref>
</weblogic-web-app>
```

5. Restart NetBeans and start Oracle WebLogic Server. Restarting NetBeans will cause the shared libraries referenced in your `weblogic.xml` file to appear in the libraries branch of your project. You can also close and re-open the project.
6. Create the Greeting resource class.
   - Right-click the HelloRest project and choose New > Other.
   - Select a category of Web Services and a file type of RESTful Web Services from Patterns.
   - Click Next.
   - Select Simple Root Resource and click Next.
   - In the Specify Resource Classes dialog box, set the following values:
     - Resource Package: `hello`
     - Path: `greeting`
     - Class Name: `GreetingResource`
     - MIME Type: `text/plain`
     - Representation Class: `java.lang.String`
     - Use Jersey-specific features (**checked**) – This will cause the `web.xml` deployment descriptor to automatically be updated with a working configuration.
   - Click Finish.
7. Modify the `GreetingResource` class to accept and return a static String value.
   - Open the `GreetingResource.java` file.
   - Add a static String variable named `text` and set it equal to `"Hello"`.

```
private static String text = "Hello";
```

   - Add a static String variable named `text` and set it equal to `"Hello"`.

```
@GET
@Produces("text/plain")
public String getText() {
    return text;
}
```

   - Add a static String variable named `text` and set it equal to `"Hello"`.

```
@PUT
@Consumes("text/plain")
public void putText(String content) {
    text = content;
}
```

- Save your changes.

8. Test the HelloRest application.
   - Deploy the HelloRest project.
   - Open the Firefox web browsers and launch the RESTClient extension.
   - Send a `GET` request to the http://localhost:7001/HelloRest/webresources/greeting URL.
   - The response headers should include a status code of 200 OK and the response body should contain:

     ```
     Hello
     ```

   - Change the HTTP Method to `PUT`.
   - Enter a new greeting message in the request body.

     ```
     Hello Dave
     ```

   - Click SEND.
   - Switch back to using the `GET` HTTP method and retrieve your updated greeting message.

     In Lesson 10: "Implementing JAX-RS Web Services" you will create more complex JAX-RS resources.

(Optional) Practice 6-5: Exploring a REST Service with cURL

Overview

In this practice, you use the cURL command-line utility to explore WebLogic RESTful Management Services.

Assumptions

You have complete Practice 6-1 to enable RESTful Management Services.

Tasks

1. Open a cmd box and change to the `D:\labs\student\resources` directory.

```
D:
cd D:\labs\student\resources
```

2. Use the cURL executable to request the list of applications running on the domain. The response should be JSON formatted.

```
curl --header "Accept: application/json" --user
weblogic:welcome1 http://localhost:7001/management/tenant-
monitoring/applications/
```

3. In a large production environment, multiple WebLogic Server instances would be installed and grouped together to form a domain. You can use WebLogic RESTful Management Services to check that a particular application is running correctly on every server (target) that it was installed to.
4. Could you design a shell script that alerted you when an application's status changed? Would JSON, XML, or HTML be easier to parse with command-line tools like sed and awk?

Practices for Lesson 7: Creating REST Clients

Chapter 7

Practices for Lesson 7: Creating REST Clients

Practices Overview

In these practices, you create multiple types of REST clients by using URLConnection, Jersey Client API, and JavaScript (jQuery).

Practice 7-1: Calling REST Services with URLConnection

Overview

In this practice, you call the WebLogic RESTful Management Services web service. The web service provides representations in three formats: HTML, JSON, and XML. Because Java includes multiple APIs to process XML formatted messages, the easiest representational format to process in Java (without additional libraries) is XML.

Assumptions

Practice 6-1: Enabling RESTful Management Services for WebLogic was completed.

Tasks

1. Start NetBeans if needed.
2. Using the Services tab, start WebLogic Service if it is not already started.
3. Create a new Java Application project named JavaSERestClient in the `D:\labs\student\exercises\lab07` folder.
   - From the NetBeans menu choose File > New Project.
   - Choose a Category of Java.
   - Choose a Project type of Java Application.
   - ClickNext.
   - Change the Project Name to JavaSERestClient.
   - Change the Project Location to `D:\labs\student\exercises\lab07`.
   - Click Finish.
4. Create a new JAXB Binding for the `D:\labs\student\exercises\lab07\monitoring.xsd` file.
   - Commonly, a RESTful web service may not provide an XML schema document even if it returns XML representations of resources. To process the XML data in Java, you can either use a processing method such as StAX or you can create JAXB annotated classes. You can either create the JAXB annotated classes by hand, and never have a XML schema file, or you can create a XML schema file by hand for the purpose of generating JAXB artifacts. For this practice, an XML schema file has been created in advance for you.
   - Right-click the JavaSERestClient project and choose New > Other.
   - Choose a Category of XML.
   - Choose a File Type of JAXB Binding.
   - Specify a Binding Name of MonitorBinding.
   - Specify a Schema File Location of `D:/labs/student/exercises/lab07/monitoring.xsd`
   - Specify a Package Name of wlsmon.
   - Click Finish.
5. Generate the JAXB classes.
   - The xjc Ant task does not run when you execute the project or run a file in it. There are two ways you can execute the xjc task to generate the JAXB classes.
     - Right-click the JavaSERestClient project and choose Build.

- Right-click the JAXB Bindings node under the JavaSERestClient project and choose Regenerate Java Code.

6. Read the http://localhost:7001/management/tenant-monitoring/applications/ resource in as an XML string.
   - Open the `javaserestclient.JavaSERestClient.java` file.
   - Below the `main` method add a `printObjectType` method that will recursively call itself and print out all child elements. You can copy/paste this method from the `D:\labs\student\exercises\lab07\printObjectType.txt` file. This method uses the JAXB generated classes to read and display the information contained in the XML response.

```
public static void printObjectType(ObjectType obj, int
indentLevel) {
    List<PropertyType> properties = obj.getProperty();
    if (properties != null) {
        String indentStr = "";
        for (int i = 0; i < indentLevel; i++) {
            indentStr += "    ";
        }
        for(PropertyType pType : properties) {
            System.out.print(indentStr + pType.getName());
            if(pType.getValue() != null) {
                ValueType valType = pType.getValue().getValue();
                System.out.println("=" + valType.getValue());
            } else {
                System.out.println("");
            }
            ArrayType arrType = pType.getArray();
            if(arrType != null) {
                List<ObjectType> objList = arrType.getObject();
                for(ObjectType obj2 : objList) {
                    printObjectType(obj2, indentLevel + 1);
                }
            }
            ObjectType objType = pType.getObject();
            if(objType != null) {
                printObjectType(objType, indentLevel + 1);
            }
        }
        System.out.println("");
    }
}
```

   - In the `main` method create a `try` block. All remaining tasks in step 6 take place inside this try block.

```
try {

} catch (IOException | JAXBException ex) {
    ex.printStackTrace();
}
```

- The WebLogic RESTful Management Service resources are protected by a username and password. Create an Authenticator that will be automatically used by HttpURLConnection.

```
Authenticator authinstance = new Authenticator() {
    @Override
    public PasswordAuthentication getPasswordAuthentication() {
        return new PasswordAuthentication("weblogic",
"welcome1".toCharArray());
    }
};
Authenticator.setDefault(authinstance);
```

- Create a connection to the http://localhost:7001/management/tenant-monitoring/applications/ resource making sure to accept only application/xml responses.

```
URL url = new URL("http://localhost:7001/management/tenant-
monitoring/applications/");
HttpURLConnection connection = (HttpURLConnection)
url.openConnection();
connection.setRequestProperty("Accept", "application/xml");
```

- If the response code was in the 400 or 500 range, then an error occurred. Print the response body so you can see what the error was.

```
if (connection.getResponseCode() >= 400) {
    InputStream rIn = connection.getErrorStream();
    BufferedReader buff = new BufferedReader(new
InputStreamReader(rIn));
    String inputLine;
    while ((inputLine = buff.readLine()) != null) {
        System.out.println(inputLine);
    }
}
```

- If the response code was below 400, then the request succeeded. Convert the response body into a single String object by using `StringBuilder`.

```
else {
    InputStream rIn = connection.getInputStream();
    BufferedReader buff = new BufferedReader(new
InputStreamReader(rIn));
    StringBuilder sb = new StringBuilder();
    String inputLine;
    while ((inputLine = buff.readLine()) != null) {
```

```
            sb.append(inputLine).append("\n");
        }
```

- Use JAXB to read the root data element of the response body and pass the child object element to the `printObjectType` method.

```
        JAXBContext jc = JAXBContext.newInstance("wlsmon");
        Unmarshaller u = jc.createUnmarshaller();

        StringReader reader = new StringReader(sb.toString());
        JAXBElement<DataType> jbDataType = (JAXBElement<DataType>)
u.unmarshal(reader);
        DataType data = jbDataType.getValue();
        printObjectType(data.getObject(), 0);
}
```

7. Run the JavaSERestClient project and correct any errors that occur.

Practice 7-2: Using the Jersey Client API

Overview

In this practice, you modify the JavaSERestClient project to use the Jersey Client API. The Jersey Client API is not a standard part of the SE or EE platform libraries; however, it is expected that the upcoming Java EE 7 REST client API will be based on the Jersey Client API.

Assumptions

You have completed Practice 7-1.

Tasks

1. Add the Jersey 1.17 library to the JavaSERestClient project.
   - You created the Jersey 1.17 library in Practice 6-3: Updating Jersey (JAX-RS).
   - Right-click the Libraries node of the JavaSERestClient project and choose Add Library.
   - Select Jersey 1.17 and click Add Library.
2. Comment out the HttpURLConnection-related code in the JavaSERestClient `main` method.
   - Comment out all the code within the main method. The Jersey Client API delegates to HttpURLConnection so the same authentication mechanism is still used. Below the commented code, re-create the Authenticator code:

```
Authenticator authinstance = new Authenticator() {
    @Override
    public PasswordAuthentication getPasswordAuthentication() {
        return new PasswordAuthentication("weblogic",
"welcome1".toCharArray());
    }
};
Authenticator.setDefault(authinstance);
```

3. Create a Jersey Client object and WebResource using the http://localhost:7001/management/tenant-monitoring URL.

```
Client client = Client.create();
String baseUrl = "http://localhost:7001/management/tenant-
monitoring";
WebResource webResource = client.resource(baseUrl);
```

   - By using a base URL that is common across all resources provided by REST service, you can reuse the WebResource.
4. Issue a `GET` request to the http://localhost:7001/management/tenant-monitoring/applications URL. Accept only application/xml responses and convert the response to a JAXB bound `DataType` class. Finally, call the `printObjectType` method to display the result.

```
try {
    DataType dataType = webResource
            .path("applications")
            .accept(MediaType.APPLICATION_XML)
            .get(DataType.class);
```

```
        printObjectType(dataType.getObject(), 0);
} catch (UniformInterfaceException ex) {
    ex.printStackTrace();
}
```

- Note the method chaining on the webResource. Most methods of the `WebResource` class return a reference to the WebResource whose method was just called. The `path`, `accept`, and `get` methods are all being called on the same object.
- Note that the `get` method returns a different return type depending on the `.class` file you provide as an argument. You can pass in JAXB annotated classes: `String.class` if you want the raw text of the response body, or commonly a `ClientResponse.class` if you need detailed information about the HTTP response (cookies, status codes, and so on).
- Note that the try/catch of `UniformInterfaceException` is optional. `UniformInterfaceException` is a subclass of `RuntimeException`. It will be thrown if the response status code was greater than or equal to 300.

5. Fix your import statements.
6. Run the JavaSERestClient project and correct any errors that occur.

(Optional) Practice 7-3: Modifying a JavaScript (jQuery) REST Client

Overview

In this practice, you open an existing jQuery REST client application and modify it to display the version of Java used by WebLogic Server.

Assumptions

Practice 6-1: Enabling RESTful Management Services for WebLogic was completed.

Tasks

1. Open the `D:\labs\student\exercises\lab07\jQueryRestClient` project.
2. Open the `index.jsp` file.
3. Starting at line 15, you see:

```
$(document).ready(function() {
    // do stuff when DOM is ready
    $('#ajaxGetLink').click(function(){
        readString();
    });
});
```

   - In the jQuery `$(document).ready(..);` is used to run code after the elements of a web page have loaded.
   - This block adds a click handler to an element with the ID of `ajaxGetLink` that appears on line 46.

```
<a id="ajaxGetLink" href="javascript:void(0)">Get</a>
```

   - Because the `href` of the link is set to `javascript:void(0)` the only thing that will happen when clicking the hyperlink is that the click handler will run.
   - The click handler for the `ajaxGetLink` calls the `readString()` function.

4. On line 22 you see the `readString()` function that is called every time the hyperlink is clicked:

```
function readString() {
    console.log('reading server status');
    $.ajax({
        cache: false,
        url: '/management/tenant-monitoring/servers/' +
$('#theinput').val(),
        dataType: "json",
        accepts: {
            text: "application/json"
        },
        success: function(data, textStatus, jqXHR){
            alert('Read ' + data);
            console.log('Read: ' + data);
        },
```

```
        error: function(jqXHR, textStatus, errorThrown){
            alert('GET error: ' + textStatus + ',' +
errorThrown);
        }
    });
}
```

- The `$.ajax(..)` method call makes the HTTP request to the REST web service.
- The url is built from a static string `'/management/tenant-monitoring/servers/'` and the current value of the form input field on line 47.
- The success function runs whenever the ajax call completes correctly. Currently, it causes an alert (pop-up) box to be displayed, which shows the value of the data variable. The data variable will contain the entire response body.

5. Run the jQueryRestClient project and test the application. Use a value of myserver in the form field.
   - If prompted for a username and password use the `weblogic`/`welcome1` combination.
6. View the success method that displays the amount of heap memory currently in use by the server named in the input box.
   - Because the response to the ajax query is JSON and line 27 says to treat it as a "json" dataType (instead of "text") you can use dot notation to navigate the JavaScript object stored in the data variable. This is similar to JSP and JSF expression language (EL) navigation.

```
alert('Used ' + data.body.item.heapSizeCurrent + ' of ' +
data.body.item.heapSizeMax);
```

7. Determine the property that contains the Java version. Modify the alert method call to display the Java version.
8. Save your changes and rerun the page. Note that you must refresh the page in the browser window in order to cause the browser to see JavaScript code changes.

(Optional) Practice 7-4: Properties of a RESTful Web Service

Overview

In this practice, you discuss with your instructor and fellow classmates what the criteria for a REST web service are.

Assumptions

You don't mind a lively discussion.

Tasks

1. Look at some of Roy Fielding's writings on REST
   - See http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven and http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.
   - Performing an Internet search for RESTafarian also leads to interesting discussions on the validity of REST web service styles.
2. Are the RESTful Management Services for WebLogic hypermedia based? How many root URLs are there?
3. What would the benefit of having a single root URL be?

Practices for Lesson 8: Bottom-up JAX-WS Web Services

Chapter 8

Practices for Lesson 8: Bottom-Up JAX-WS Web Services

Practices Overview

In these practices, you create JAX-WS web service endpoints by using a code-first approach.

Practice 8-1: Creating the Card Game Service

Overview

In a previous practice, Practice 5-2: Creating a Card Deck Web Service, you created a SOAP service that allowed for the creation, listing, shuffling, and deleting of cards.

In this practice, you will create a GenericCardGameWS project. The new generic card game web service will contain the functionality needed for many card games such as:

- Combining card decks
- Dealing handing
- Creating card piles (stock pile, discard pile, and so on)

If you were creating multiple card games in Java such as Klondike (solitaire) and Indian Rummy you would place all the common functionality into a shared library JAR. With web services, you should imagine that someone is creating Klondike using C#, Indian Rummy using Java, and any number of games are being created using any possible language. You could either attempt to re-create the shared library in every programming language or you could make it available as a web service.

Creating a web service instead of a shared library does have costs; such as, the overhead of network communication and adding another potential point of failure to an application. The primary benefits of web services are platform neutrality.

If you can guarantee that you will have only a single client implementation for your service and both the client and service will be implemented in the same language, you might not need to implement your service as a web service. A shared library or network service using a platform-specific technology (like Java's RMI) might be a better choice.

In a later lesson, you will create a client for this generic card game service, but because you haven't reached that part of the course yet, you cannot be sure which programming language will be used for the client. Therefore, a web service is the safest choice for the shared card game functionality.

Assumptions

Practice 5-2: Creating a Card Deck Web Service was completed and the CardDecksWS WAR is deployed to WebLogic Server.

Tasks

1. Start NetBeans.
2. Create a new web application project named GenericCardGameWS in the `D:\labs\student\exercises\lab08` folder.
   - From the NetBeans menu choose File > New Project.
   - Choose a Category of Java Web.
   - Choose a Project type of Web Application.
   - Click Next.
   - Change the Project Name to GenericCardGameWS.
   - Change the Project Location to `D:\labs\student\exercises\lab08`.
   - Click Next
   - Deselect Set Source Level to 6.
   - Click Finish.
3. Change the Source/Binary Format to JDK7 for the GenericCardGameWS project.

- WebLogic Service 12*c* is certified to run on JDK7 and you will use Java 7 features.
- Right-click the GenericCardGameWS project and choose Properties.
- In the Sources category change the Source/Binary Format to JDK7.
- Click OK.

4. The GenericCardGameWS project will provide a web service, but it will also be a client of the CardDecksWS service. Add a new Web Service Client to the GenericCardGameWS project.
   - Right-click the GenericCardGameWS project and choose New > Other.
   - Select a category of Web Services and a file type of Web Service Client.
   - Click Next.
   - Select WSDL URL and enter http://localhost:7001/CardDeckSessionBean/CardDeckSessionBeanService?wsdl.
   - Leave the package name empty.
   - Click Finish.
   - Under Generated Sources, the WSDL-related classes should appear in an ejb package and the JAXB domain classes should appear in dukesdecks. A package named ejb runs the risk of causing conflict with classes that you create. Change the package name for the generated WSDL classes.
   - Expand Web Service References.
   - Right-click CardDeckSessionBeanService and choose Edit Web Service Attributes.
   - Switch to the Wsimport Options tab.
   - Add a new Wsimport Option named `package` with a value of `deckws`.
   - Click the OK button.
   - Right-click CardDeckSessionBeanService and choose Refresh. Click the Yes button in the Confirm Client Refresh dialog box that appears to confirm the client refresh.
5. Create the CardGame POJO.
   - Right-click the GenericCardGameWS project and choose New > Other.
   - Select a category of Java and a file type of Java Class.
   - Click Next.
   - Set the Class Name to CardGame.
   - Set the Package name to “games”.
   - Click Finish.
6. The CardGame POJO should be capable of representing the card layout of most card games. A card game starts off with one or more card decks, cards can be split into piles, shuffled, and a card(s) can be drawn from a pile. A pile can be something like the cards in a player’s hand, a discard pile, or a stock pile. Even single-player games like Klondike (solitaire) use piles such as foundation piles, downturn piles, and so on. Effectively, pile management forms the basis of any card game. Add pile management code to the `CardGame` class.
   - Add two fields to the `CardGame` class. A `List<CardType> supply` field is where all the cards are stored when creating a game, it serves as the source of cards when creating new piles in a game. A `Map<String, List<CardType>>` piles field is used to keep track of all the piles within a game. All piles have unique names.

```
private List<CardType> supply = new ArrayList<>();
private Map<String, List<CardType>> piles = new HashMap<>();
```

- Create an `addCards(List<CardType> deckCards)` method that is used to add cards to the game.

```
public void addCards(List<CardType> deckCards) {
    supply.addAll(deckCards);
}
```

- Create a `createPile(int pileSize, String pileName)` method that is used to create a pile of named cards using the `supply` cards as the source.

```
public void createPile(int pileSize, String pileName) {
    List<CardType> list = supply.subList(0, pileSize);
    piles.put(pileName, new ArrayList(list));
    list.clear();
}
```

- Add an inner enum named `DistributionMethod` that will be used to indicate how cards are dealt when creating multiple named piles in a single operation.

```
public enum DistributionMethod {
    ROUND_ROBIN,
    FILL_PILE;
}
```

- Create a `createPiles(int pileSize, DistributionMethod dist, String... pileNames)` method that is used to create multiple named piles of cards using the `supply` cards as the source. Use the `dist` parameter to determine how cards are distributed from the supply cards to the named piles.

```
public void createPiles(int pileSize, DistributionMethod dist,
String... pileNames) {
    for (String pileName : pileNames) {
        piles.put(pileName, new ArrayList());
    }
    switch (dist) {
        case ROUND_ROBIN:
            for (int i = 0; i < pileSize; i++) {
                for (String pileName : pileNames) {
                    piles.get(pileName).add(supply.remove(0));
                }
            }
            break;
        case FILL_PILE:
            for (String pileName : pileNames) {
                for (int i = 0; i < pileSize; i++) {
                    piles.get(pileName).add(supply.remove(0));
                }
            }
            break;
```

```
    }
}
```

- Create two methods, `showTopCard(String pileName)` and `showPile(String pileName)`, which can be used to retrieve a single card or all cards from a named pile.

```
public CardType showTopCard(String pileName) {
    return piles.get(pileName).get(0);
}
```

```
public List<CardType> showPile(String pileName) {
    return piles.get(pileName);
}
```

- Create a `transferCards(String sourcePile, String destinationPile, int cardCount)` method that removes cards from one named pile and adds them to another named pile.

```
public void transferCards(String sourcePile, String
destinationPile, int cardCount) {
    List<CardType> list = piles.get(sourcePile).subList(0,
cardCount);
    piles.get(destinationPile).addAll(list);
    list.clear();
}
```

- Create a private `shuffleCardList(List<CardType> cardList)` method that takes a list of cards and shuffles them. Having a good, non-predictable shuffling method is important; the code below uses the Durstenfeld algorithm that is based on the Fisher-Yates method.

```
private void shuffleCardList(List<CardType> cardList) {
    ThreadLocalRandom tlRandom = ThreadLocalRandom.current();
    for (int topIndex = cardList.size() - 1; topIndex > 0;
topIndex--) {
        int randIndex = tlRandom.nextInt(topIndex + 1);
        CardType tempCard = cardList.get(randIndex);
        cardList.set(randIndex, cardList.get(topIndex));
        cardList.set(topIndex, tempCard);
    }
}
```

- Create a public `shufflePile(String pileName)` method that allows a named pile to be shuffled.

```
public void shufflePile(String pileName) {
    shuffleCardList(piles.get(pileName));
}
```

- Create a public `shuffleAll()` method that transfers all the piles back to the `supply` cards and shuffles them.

```
public void shuffleAll() {
    for (List<CardType> pile : piles.values()) {
        supply.addAll(pile);
        pile.clear();
    }
    piles.clear();
    shuffleCardList(supply);
}
```

7. Create the GameStorageBean Singleton EJB.
   - While you can easily annotate some types of EJBs to make them available as a web service, it is more common to create a POJO endpoint and delegate to an EJB if you need the features that an EJB provides. Here, you create an EJB to provide thread-safe persistence (in-memory).
   - Right-click the GenericCardGameWS project and choose New > Other.
   - Select a category of Enterprise JavaBeans and a file type of Session Bean.
   - Click Next.
   - Set the EJB Name to GameStorageBean.
   - Set the Package name to “games”.
   - Change the Session Type to Singleton.
   - Click Finish.
8. Complete the GameStorageBean. This Singleton EJB stores all card games in-memory using a `Map<String,CardGame> games` field.

```
private Map<String,CardGame> games = new HashMap<>();
```

   - Create a method for retrieving a CardGame from the Singleton bean.

```
public CardGame getGame(String gameId) {
    return games.get(gameId);
}
```

   - Create a method to add a CardGame. The Singleton EJB should assign a unique string ID to each new game and return it.

```
public String addGame(CardGame game) {
    String id = UUID.randomUUID().toString();
    games.put(id, game);
    return id;
}
```

   - Create a method to remove a game from the games map. Return `true` if a game was deleted.

```
public boolean removeGame(String gameId) {
    CardGame game = games.remove(gameId);
    if(game == null) {
        return false;
    } else {
        return true;
```

```
    }
}
```

9. Create the CardGameService endpoint.
   - Right-click the GenericCardGameWS project and choose New > Other.
   - Select a category of Web Services and a File Type of Web Service.
   - Click Next.
   - Set the Web Service Name to CardGameService.
   - Set the Package name to “games”.
   - Leave Create Web Service from Scratch selected.
   - Click Finish.
   - Delete the `hello` method from the `CardGameService` class.
10. Add the `createCardGame` and `removeCardGame` operations to `CardGameService`.
   - Both the `createCardGame` and `removeCardGame` operations delegate to the GameStorageBean you just created. Web service endpoint classes are often wrappers around other classes; it is very common to perform a lot of delegation in an endpoint class.
   - Add an injected reference to the GameStorageBean.

```
@EJB
private GameStorageBean bean;
```

   - Add a `removeCardGame(String gameId)` method.

```
public boolean removeCardGame(String gameId) {
    return bean.removeGame(gameId);
}
```

   - Add an empty `createCardGame` method to the `CardGameService` class.

```
public String createCardGame(int numberOfDecks,
                             int jokerCountPerDeck) {


}
```

   - To create a game, you must first create the correct number of card decks. The CardGameService uses the CardDeckSessionBeanService as a source of cards. Add information about the locally copied WSDL file for the CardDeckSessoinBean.

```
private static final URL wsdlURL;
private static final String wsdlLocation = "WEB-
INF/wsdl/localhost_7001/CardDeckSessionBean/CardDeckSessionBeanS
ervice.wsdl";

static {
    try {
        wsdlURL = new URL("file://./" + wsdlLocation);
    } catch (MalformedURLException ex) {
        throw new ExceptionInInitializerError(ex);
    }
```

```
}
```

- For this class, you should avoid using a `@WebServiceRef` to inject a service or proxy (port); resources injected with the `@WebServiceRef` annotation are not guaranteed be thread-safe. Because GameStorageBean is a POJO JAX-WS endpoint, it is multi-threaded (similar to a Servlet). In the `createCardGame` method, obtain the service and port for the CardDeckSessionBean.

```
CardDeckSessionBeanService service = new
CardDeckSessionBeanService(wsdlURL);
CardDeckSessionBean port = service.getCardDeckSessionBeanPort();
```

- Complete the `createCardGame` method by using the CardDeckSessionBean to obtain the cards to be added to a game. Delete the decks after adding their cards to a CardGame.

```
CardGame game = new CardGame();

for (int i = 0; i < numberOfDecks; i++) {
    int deckId = port.createDeck(jokerCountPerDeck);
    StackType deck = port.getDeck(deckId);
    game.addCards(deck.getCard());
    port.deleteDeck(deckId);
}

return bean.addGame(game);
```

- Add methods to the `CardGameService` class that allow clients to interact with a stored CardGame. Because of the multi-threaded nature of CardGameService you should synchronize on a game to ensure only one client at a time is modifying it. Because of the stateless natural of the CardGameService, the client has to pass in a game ID on every call. You should add the following methods:
  - `public void createPile(String gameId, int pileSize, String pileName)`
  - `public void createPiles(String gameId, int pileSize, DistributionMethod dist,  String... pileNames)`
  - `public CardType showTopCard(String gameId, String pileName)`
  - `public List<CardType> showPile(String gameId, String pileName)`
  - `public void transferCards(String gameId, String sourcePile, String destinationPile, int cardCount)`
  - `public void shuffleAll(String gameId)`
  - `public void shufflePile(String gameId, String pileName)`

  The following is an example of the `showTopCard` method.

```
public CardType showTopCard(String gameId, String pileName) {
    CardGame game = bean.getGame(gameId);
    synchronized (game) {
        return game.showTopCard(pileName);
    }
```

```
}
```

11. Build and deploy the GenericCardGameWS project.
12. Test the CardGameService web service.
    - Expand the Web Services branch.
    - Right-click CardGameService and choose Test Web Service. The WebLogic Test Client application should appear in a web browser. Alternatively, you can manually direct a web browser to http://localhost:7001/wls_utc/begin.do?wsdlUrl=http://localhost:7001/GenericCardGameWS/CardGameService?wsdl.
    - Find the `createPiles` operation. Notice that it appears different than other operations. This is because one of the parameters is an enum and an unbounded set of strings (pile names). When testing the `createPiles` operation you edit the inbound SOAP body directly.
    - Find the `createCardGame` operation. Create a new game by asking for a game with 2 decks and 1 joker per deck.
    - Start playing a game of cards. Shuffle all the cards, deal out 4 hands (pile names: hand1, hand2, hand3, hand4) of 13 cards each. Create a discard pile of 1 card and put the remaining cards into a stock pile.
    - Looking at the WebLogic Test Client application, can you tell the purpose of the two parameters required by the `createCardGame` operation? The WebLogic Test Client does not access the Java code for the tested web service. If you look at the WSDL file and included XML Schemas, do you see any additional information about the parameters of the `createCardGame` method?
      - http://localhost:7001/GenericCardGameWS/CardGameService?wsdl
      - http://localhost:7001/GenericCardGameWS/CardGameService?xsd=1
      - http://localhost:7001/GenericCardGameWS/CardGameService?xsd=2
13. Improve the generated element names. When using the WebLogic Test Client, you might have had to reference the CardGameService source to figure out what a method parameter was for because the names are non-descriptive. Use the `@WebParam` annotation to fix that.
    - Modify the `removeCardGame` method parameter to include a `@WebParam` annotation.

```
public boolean removeCardGame(@WebParam(name="game-id") String
gameId) {
    return bean.removeGame(gameId);
}
```

    - Notice the naming convention used by the `@WebParam` example. Java developers typically use very long descriptive names with camel case. While there is no one true standard, one of the conventions that is popular for XML is the use of all lowercase names separated by hyphens.
    - Save you changes and reload the WebLogic Test Client application. You should notice a more descriptive name for the parameter of the `removeCardGame` operation. You should also see the changes in the WSDL and included XML Schema files.
      - http://localhost:7001/GenericCardGameWS/CardGameService?wsdl
      - http://localhost:7001/GenericCardGameWS/CardGameService?xsd=1
      - http://localhost:7001/GenericCardGameWS/CardGameService?xsd=2
    - Add `@WebParam` annotations to all method parameters in the `CardGameService` class.

```
public String createCardGame(@WebParam(name="deck-count") int
numberOfDecks,
        @WebParam(name="jokers-per-deck") int jokerCountPerDeck)
{
```

```
public boolean removeCardGame(@WebParam(name="game-id") String
gameId) {
```

```
public void createPile(@WebParam(name="game-id") String gameId,
        @WebParam(name="size") int pileSize,
        @WebParam(name="pile") String pileName) {
```

```
public void createPiles(@WebParam(name="game-id") String gameId,
        @WebParam(name="size") int pileSize,
        @WebParam(name="dist-method") DistributionMethod dist,
        @WebParam(name="pile") String... pileNames) {
```

```
public CardType showTopCard(@WebParam(name="game-id") String
gameId,
        @WebParam(name="pile") String pileName) {
```

```
public List<CardType> showPile(@WebParam(name="game-id") String
gameId,
        @WebParam(name="pile") String pileName) {
    CardGame game = bean.getGame(gameId);
    synchronized (game) {
```

```
public void transferCards(@WebParam(name="game-id") String
gameId,
        @WebParam(name="source-pile") String sourcePile,
        @WebParam(name="dest-pile")  String destinationPile,
        @WebParam(name="cards") int cardCount) {
```

```
public void shuffleAll(@WebParam(name="game-id") String gameId)
{
    CardGame game = bean.getGame(gameId);
    synchronized (game) {
```

```
public void shufflePile(@WebParam(name="game-id") String gameId,
        @WebParam(name="pile-name") String pileName) {
```

- Save your changes and reload the WebLogic Test Client application. You should also see the changes in the WSDL and included XML Schema files.

- http://localhost:7001/GenericCardGameWS/CardGameService?wsdl
- http://localhost:7001/GenericCardGameWS/CardGameService?xsd=1
- http://localhost:7001/GenericCardGameWS/CardGameService?xsd=2

14. If you attempt to play a game (see the next two bullet points that are repeated from an earlier tasks), you will find that one source of difficulty is the placement of all remaining cards into a pile. Currently, the client has to keep track of the number of remaining cards.
    - Find the `createCardGame` operation. Create a new game by asking for a game with 2 decks and 1 joker per deck.
    - Start playing a game of cards. Shuffle all the cards, deal out 4 hands (pile names: hand1, hand2, hand3, hand4) of 13 cards each. Create a discard pile of 1 card and put the remaining cards into a stock pile.
15. Add an overloaded `createPile` method in the `CardGameService` class that doesn't need to know the number of card to add to a pile; it takes all the remaining supply cards.
    - Start by modifying the `CardGame` class to allow the server to see how many cards are in a pile or remain in the supply.

```
public int getSupplySize() {
    return supply.size();
}

public int getPileSize(String pileName) {
    List<CardType> pile = piles.get(pileName);
    if(pile != null) {
        return pile.size();
    } else {
        return 0;
    }
}
```

    - Switch back to the `CardGameService` class and add an overloaded `createPile` method.

```
public void createPile(@WebParam(name = "game-id") String
gameId,
        @WebParam(name = "pile") String pileName) {
    CardGame game = bean.getGame(gameId);
    synchronized (game) {
        game.createPile(game.getSupplySize(), pileName);
    }
}
```

    - Save, deploy, and test your application. What is the result?
    - If you look in the Oracle WebLogic Server Output tab in NetBeans you should see some stacktraces. The root cause is that you have two operations with the same name. Just because Java supports method overloading doesn't mean that every client will. The simplest solution is to rename the method but if you don't want to do that…
    - Add the following annotations to the new `createPile` method.

```
@WebMethod(operationName = "create-all-supply-pile")
```

```
@ResponseWrapper(className = "CreateAllSupplyPileResponse")
@RequestWrapper(className = "CreateAllSupplyPile")
```

- A service endpoint method is mapped to a XML tag name (the first tag inside the SOAP body in document/literial/wrapped type). To parse and generate the content inside the SOAP body element, a pair of JAXB classes are generated, the names of these generated classes are `<MethodName>` and `<MethodName>Response`. To avoid conflicts, you must change the tag that is used (via `@WebMethod(operationName)`) and you must change the names of the generated classes used to marshall/unmarshall the SOAP body.

(Optional) Practice 8-2: Publishing Endpoints Without an Application Server

Overview

In this practice, you publish a simple endpoint by using only Java SE.

Tasks

Tasks

1. Start NetBeans.
2. Create a new Java application project named JavaSEWebService in the `D:\labs\student\exercises\lab08` folder.
   - From the NetBeans menu choose File > New Project.
   - Choose a Category of Java.
   - Choose a Project type of Java Application.
   - Click Next.
   - Change the Project Name to JavaSEWebService.
   - Change the Project Location to `D:\labs\student\exercises\lab08`.
   - Click Finish.
3. Create the HelloService endpoint.
   - Right-click the JavaSEWebService project and choose New > Other.
   - Select a Category of Java and a File Type of Java Class.
   - Click Next.
   - Set the Web Service Name to HelloService.
   - Set the Package name to javasewebservice.
   - Click Finish.
4. Add a `getMessage` method to the `HelloService` class.

```
    public String getMessage(
            @WebParam(name = "name") String name) {
        return name + " rocks!";
    }
```

5. Add a `@WebService` annotation to the `HelloService` class.
   - Add any needed imports.
6. Publish the endpoint using the JavaSEWebService `main` method.
   - Open the `JavaSEWebService.java` file and add the code to publish HelloService.

```
Endpoint endPoint =
Endpoint.publish("http://localhost:8080/Hello", new
HelloService());
```

7. Run the JavaSEWebService project.
8. Open the Firefox web browser and visit http://localhost:8080/Hello.
   - You should see a summary of the web service.
   - Using the RESTClient Firefox extension, view the WSDL for the service.

- You should see an imported XML Schema in the WSDL; open that URL in a new RESTClient tab. (Some of the XML information like targetNamespace is not shown by a regular browser tab.)

9. Test HelloService by using the RESTClient Firefox extension.
   - Use a service URL of http://localhost:8080/Hello.
   - Change the Method to `POST`.
   - Add a Content-Type header of text/xml.
   - Craft a SOAP request message based on the WSDL and XML Schema of the service.

```
<env:Envelope
xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
  <env:Header />
  <env:Body>
  <getMessage xmlns="http://javasewebservice/">
    <name xmlns="">Tom</name>
  </getMessage>
  </env:Body>
</env:Envelope>
```

   - Send the request. You should receive a response that resembles:

```
<?xml version="1.0" ?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <ns2:getMessageResponse
xmlns:ns2="http://javasewebservice/">
      <return>Tom rocks!</return>
    </ns2:getMessageResponse>
  </S:Body>
</S:Envelope>
```

10. Switch back to NetBeans and stop the service.
    - In the Output tab you should see a red square button; this is the stop button that is read only when a process is running. Along the very bottom border of NetBeans window you should see a summary of running processes (with an X to cancel them).
    - It is very common to forget to close a running process. The results can be minimal or cause errors depending on the application.
    - Without stopping the first run, try to run a second instance of the JavaSEWebService project. You should get an exception because only one application can listen on a network port at a time.
    - Stop the first instance of the application.
11. Endpoint.publish uses a small web server that is built in to the Java SE runtime classes. You can customize the behavior of the server (change the number of threads, configure https, and so on) if desired.
    - For more information see http://docs.oracle.com/javase/7/docs/jre/api/net/httpserver/spec/com/sun/net/httpserver/package-summary.html.

Practices for Lesson 9: Top-down JAX-WS Web Services

Chapter 9

Practices for Lesson 9: Top-Down JAX-WS Web Services

Practices Overview

In these practices, you create JAX-WS web service endpoints by using a WSDL-first approach.

Practice 9-1: Creating the Player Management Service

Overview

In a previous practice, Practice 8-1: Creating the Card Game Service, you created a SOAP service that provides the building blocks of card games (pile management).

In this practice, you will create a PlayerManagementService. By itself the CardGameService only keeps track of the cards. Something needs to keep track of the players, which players are in a game, the win/loss record, which hand (pile) belongs to which player, and so on. The PlayerManagementService will keep track of all this information.

For this practice, you will create the PlayerManagementService with a top-down (WSDL first) approach. Using a bottom-up (code first) approach as seen in Practice 8-1 is typically the preferred approach and is easier to realize for Java developers. However, sometimes you must start with a WSDL first. The most common reason is when re-implementing an existing web service in a new programming language.

When using a top-down approach, the WSDL file that is used to implement the Java-based web service must conform to the features that are provided by your JAX-WS stack. For example, the service should use the document/literal/wrapped style and be WS-Basic compliant.

For this practice, you will be supplied with an EJB that can manage users and groups that are stored in WebLogic Server. Java EE application servers provide standard mechanisms to restrict resource access and read principal, role, and group information, but have no standard way to provision users and groups. WebLogic Server uses an internal LDAP server to store user information by default. Normally, to manage users and groups in this LDAP server, you must go to the WebLogic Administration Console. The provided EJB uses the JMX MBeans provided by WebLogic Server to create an application-level programming interface. In this practice, you will use the LDAP server as a simple data store; later practices will read the information placed in the LDAP server to perform authentication and authorization by using standard Java EE security mechanisms.

Assumptions

Practice 8-1: Creating the Card Game Service was completed.

Tasks

1. Start NetBeans.
2. Create the BEA Utils server library.
   - The provided user management EJB uses classes that are available when running in WebLogic Server but are not typically included in a NetBeans project.
   - Open the Tools > Ant Libraries menu item.
   - Click the New Library button.
   - The Library Name should be BEA Utils.
   - The Library Type should be Server Libraries.
   - Click OK.
   - While the BEA Utils library is selected, click the Add JAR/Folder button.
   - Select the `D:\weblogic\wls\modules\com.bea.core.utils_2.0.0.0.jar` file and click the Add JAR/Folder button.
   - ClickOK to close the Ant Library Manager.
3. Open the `D:\labs\student\exercises\lab09\UserManager` project.
4. Run the UserManager project.

- Right-click the UserManager project and select Deploy.
- Using a web browser, go to http://localhost:7001/UserManager/WLSAuthTestServlet.
- You should see a list of users and groups.

5. View the users and groups with the WebLogic Server Administration Console.
   - Using a web browser, go to http://localhost:7001/console/.
   - Log in with a username of `weblogic` and a password of `welcome1`.
   - On the left, in the Domain Structure box, click Security Realms.
   - Click the myrealm realm.
   - Open the Users and Groups tab.
   - Explore the users and groups that exist; they should match those displayed by the WLSAuthTestServlet.
   - Note that both users and groups have a description field that can be used to store any text data.
6. View the WLSAuthTestServlet.
   - Switch back to NetBeans.
   - In the UserManager project, open the WLSAuthTestServlet.
   - Around line 26 you should see the WLSJMXManagementSessionBean being injected.

```
@EJB
private WLSJMXManagementSessionBean bean;
```

   - Around lines 51-74 you should see where the servlet uses the WLSJMXManagementSessionBean to obtain a list of users and their groups along with a listing of groups and their members.

```
try {
    out.println("<h2>Users</h2>");
    List<String> users = bean.getUserList();
    for(String userName : users) {
        out.print("User: " + userName + " - Groups: ");
        List<String> groups = bean.getMemberGroups(userName);
        for(String group : groups) {
            out.println(group + " ");
        }
        out.println("<br/>");
    }
    out.println("<h2>Groups</h2>");
    List<String> groups = bean.getGroupList();
    for(String groupName : groups) {
        out.print("Group: " + groupName + " - Users: ");
        List<String> groupMembers =
bean.getGroupMembers(groupName);
        for(String user : groupMembers) {
            out.println(user + " ");
        }
        out.println("<br/>");
```

```
    }
} catch (Exception ex) {
    out.println("Failed to list users and groups: " +
ex.getMessage());
}
```

7. Explore the WLSJMXManagementSessionBean EJB.
   - JMX programming and the WebLogic-specific security MBeans are beyond the scope of this course. For more on these topics see:
     - http://docs.oracle.com/javase/tutorial/jmx/index.html
     - http://docs.oracle.com/cd/E24329_01/web.1211/e24415/toc.htm
     - http://docs.oracle.com/cd/E24329_01/apirefs.1211/e24395/index.html
   - Identify the public methods of the WLSJMXManagementSessionBean EJB; these are the methods used to programmatically manage users and groups.
   - Find the methods that modify or delete a user or group. You should see that the EJB is hard-coded to prevent the modification or deletion of accounts that are typically in place in a WebLogic Server.
8. View the `D:\labs\student\exercises\lab09\UserManagement.wsdl` WSDL file.
   - Switch to the Favorites tab and navigate to the `D:\labs\student\exercises\lab09` folder to view the WSDL file.
   - This file could have been used by a web service implementation in any programming language.
   - Switch back to the Project tab.
9. Create the PlayerManagementService by using an existing WSDL file.
   - Right-click the UserManager project and choose New > Other.
   - Select a category of Web Services and a file type of Web Service from WSDL.
   - Click Next.
   - Set the Web Service Name to PlayerManagementService.
   - Set the Package name to auth.
   - Set the WSDL location to `D:\labs\student\exercises\lab09\UserManagement.wsdl`.
   - Click Finish.
10. The `auth/PlayerManagementService.java` file should automatically be generated along with several classes in the auth package under Generated Sources.
11. Complete the `PlayerManagementService` class.
    - Open the `PlayermanagementService.java` file using the Source view.
    - Notice how the operation and parameter name were automatically converted from all-lowercase-with-hyphens to javaCamelCase.
    - Add a java.util.Logger reference to the class.

```
private static final Logger logger =
Logger.getLogger(PlayerManagementService.class.getPackage().getN
ame());
```

    - Add an injected reference to a EJB of type WLSJMXManagementSessionBean.

```
@EJB
```

```
private WLSJMXManagementSessionBean bean;
```

- Implement the `createGroup` method. Delegate to the WLSJMXManagementSessionBean and catch any thrown exceptions and log them.

```
public void createGroup(java.lang.String groupName,
java.lang.String description) {
    try {
        bean.createGroup(groupName, description);
    } catch (Exception ex) {
        logger.log(Level.SEVERE, "Failed to create group", ex);
    }
}
```

- Complete the remaining methods in the `PlayerManagementService` class. For methods that must return a value, if an exception is thrown: after logging a message, return "empty" data such as a `false` value for Booleans, `""` for strings, or new `ArrayList<>()` for `List<String>`.
- Some method names in the web service may have slightly different names than in the EJB; use the method name that implies similar functionality. There should be a one-to-one mapping of methods in the web service to EJB methods.
- Add any needed imports and save all your changes.

12. Clean, build, and deploy your project.
13. Test the user-management-service.
    - Expand the Web Services branch of the project.
    - Right-click user-management-service and choose Test Web Service. You can also open a web browser and visit http://localhost:7001/wls_utc/begin.do?wsdlUrl=http://localhost:7001/UserManager/user-management-service?wsdl.
    - Using the WebLogic Test Client, try the `list-users` and `list-groups` operations.
    - Try the `create-user` operation with the following values:
        - user-name: `bob`
        - password: `welcome`
        - description: `robert`
    - Invoke the `list-users` operation. Is the user `bob` present?
    - Switch back to NetBeans and look on the Output > Oracle WebLogic Server tab. You should see a log message and stack trace.
    - The problem is that WebLogic Server enforces password complexity requirements by default. All passwords must be at least eight characters and contain one special character such as a symbol or number.
    - Create the `bob` user, but this time use a password of `welcome1`.
    - List all users; you should see the `bob` user now.
    - Test all the operations to ensure they work correctly. After you finish testing, use the `delete-*` operations to clean up all the test data.
14. There are two problems with this web service currently:
    - Clients have no idea if an exception occurs on the server side. A later practice will explore how exceptions are handled by web services.

- Security is nonexistent. Anyone that can connect to the web service can modify or delete almost any information (except the values hard-coded to be non-editable in the EJB). A later practice will look at securing web services.

Practices for Lesson 10: Implementing JAX-RS Web Services

Chapter 10

Practices for Lesson 10: Implementing JAX-RS Web Services

Practices Overview

In these practices, you create a RESTful web service that can be used to play Indian Rummy (also known as 13 Cards Rummy).

Practice 10-1: The Rules of Indian Rummy

Overview

In this practice you read the rules of playing a game of Indian Rummy.

Assumptions

You have played a card game.

Tasks

1. Using NetBeans, open the D:\labs\student\projects\RummyRules project.
2. Run the RummyRules project.
3. Read the rules for playing Indian Rummy.
   - Do not worry about all the variations, they are optional and only there for completeness (and for people who finish their practices early).
4. If you mapped a game of Indian Rummy to the GenericCardGameWS web service that splits a card game into named card piles, how many piles would you have and what would their names be?

Practice 10-2: Creating the Indian Rummy Web Service Project

Overview

In this practice, you create the IndianRummyWS project that will contain RESTful resources for playing a game of Indian Rummy. The Indian Rummy service will leverage the GenericCardGameWS and PlayerManagementService web services as building blocks. Effectively you are creating a REST web service, which is a mix of other web services with game-specific rules and resource representations.

You will create the root resource and leverage a supplied resource to represent players. In practice 10-4 you will create the resources that represent games of Indian Rummy.

Assumptions

You have completed Practice 8-1: Creating the Card Game Service and Practice 9-1: Creating the Player Management Service.

Tasks

1. Start NetBeans.
2. Create a new Web Application project named IndianRummyWS in the D:\labs\student\exercises\lab10 folder.
   - From the NetBeans menu choose File > New Project.
   - Choose a Category of Java Web.
   - Choose a Project type of Web Application.
   - Click Next.
   - Change the Project Name to IndianRummyWS.
   - Change the Project Location to D:\labs\student\exercises\lab10.
   - Click Next.
   - Deselect Set Source Level to 6.
   - Click Finish.
3. Change the Source/Binary Format to JDK7 for the IndianRummyWS project.
   - WebLogic Service 12*c* is certified to run on JDK7 and you will use Java 7 features.
   - Right-click the IndianRummyWS project and choose Properties.
   - In the Sources category, change the Source/Binary Format to JDK7.
   - Click OK.
4. Add the shared library reference for Jersey 1.17.
   - In the `weblogic.xml` configuration file, add a library-ref element that instructs WebLogic Server to use the previously installed shared library when running this application.
   - Open the `weblogic.xml` file (under Configuration Files) and add the library reference.

```
<?xml version="1.0" encoding="UTF-8"?>
<weblogic-web-app
xmlns="http://xmlns.oracle.com/weblogic/weblogic-web-app"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd
http://xmlns.oracle.com/weblogic/weblogic-web-app
```

```
http://xmlns.oracle.com/weblogic/weblogic-web-app/1.0/weblogic-
web-app.xsd">
    <jsp-descriptor>
        <keepgenerated>true</keepgenerated>
        <debug>true</debug>
    </jsp-descriptor>
    <context-root>/IndianRummyWS</context-root>
    <library-ref>
        <library-name>Jersey-1.17</library-name>
        <specification-version>1.17</specification-version>
        <implementation-version>1.17.0</implementation-version>
        <exact-match>true</exact-match>
    </library-ref>
</weblogic-web-app>
```

5. Restart NetBeans and start Oracle WebLogic Server. Restarting NetBeans will cause the shared libraries referenced in your `weblogic.xml` file to appear in the libraries branch of your project. You can also close and reopen the project.
6. The IndianRummyWS project will provide a web service but it will also be a client of the GenericCardGameWS and PlayerManagementService web services. Add two new Web Service Clients to the IndianRummyWS project.
   - Right-click the IndianRummyWS project and choose New > Other.
   - Select a category of Web Services and a file type of Web Service Client.
   - Click Next.
   - Select WSDL URL and enter http://localhost:7001/GenericCardGameWS/CardGameService?wsdl.
   - Enter a package name of cards.
   - Click Finish.
   - Right-click the IndianRummyWS project and choose New > Other.
   - Select a category of Web Services and a file type of Web Service Client.
   - Click Next.
   - Select WSDL URL and enter http://localhost:7001/UserManager/user-management-service?wsdl.
   - Enter a package name of users.
   - Click Finish.
7. Clean and build the project to generate the JAX-WS client artifacts.
8. Add D:\labs\student\exercises\lab10\CardUtils.java to the project.
   - Create a new `util` package.
   - Using the Favorites tab in NetBeans copy the D:\labs\student\exercises\lab10\CardUtils.java file.
   - Using the Projects tab in NetBeans paste the CardUtils.java file to the util package of the IndianRummyWS project.
   - The Card class in this project (most likely called CardType) was generated by wsimport based solely on the WSDL file and any referenced XML Schemas. The client-side

CardType class is lacking any business logic that was in the Card class internal to the web service.

- The CardUtils class provides methods to aid in the handling of cards. If your generated JAX-WS client code is different because of WSDL differences you may have to adjust the CardUtils source code.

9. Create the JAXB classes used by the application.
   - JAXB classes will be used to create both XML and JSON representation of resources.
   - Create the `rummy.jaxb.PlayerStats` class.
   - Create a plain Java class in the `rummy.jaxb` package named PlayerStats.
     - Add two fields, wins and losses, which are required XML elements of type public int.

```
@XmlElement(required = true)
public int wins;
@XmlElement(required = true)
public int losses;
```

     - Annotate the class with JAXB annotations. Remember that without `@XmlAccessorType` the default behavior is to serialize all public members (fields and methods).

```
@XmlRootElement(name="stats")
@XmlType(propOrder={"wins", "losses"})
public class PlayerStats {
```

     - Fix any import statements.
   - Create the `rummy.jaxb.Hand` class.
     - Create a plain Java class in the `rummy.jaxb` package named Hand.
     - Add a String href property. Right-click the source window and choose Insert Code > Add Property.
     - Add a List<CardType> cards property.
     - Using @XmlElement make the href property required.

```
@XmlElement(required=true)
public String getHref() {
    return href;
}
```

     - Annotate the class with JAXB annotations.

```
@XmlRootElement
@XmlType(propOrder = {"href", "cards"})
public class Hand {
```

   - Create the `rummy.jaxb.Player` class.
     - Create a plain Java class in the `rummy.jaxb` package named Player.
     - Add a String userName property. Right-click the source window and choose Insert Code > Add Property.
     - Add a PlayerStats stats property.
     - Add a String password property.
     - Add a String href property.

- Add a Hand hand property.
- Add a toString method when returns the userName value.

```
@Override
public String toString() {
    return userName;
}
```

- Using @XmlElement make the userName property required.

```
@XmlElement(required=true)
public String getUserName() {
    return userName;
}
```

- Annotation the class with JAXB annotations.

```
@XmlRootElement
@XmlType(propOrder={"userName", "stats", "password", "href",
"hand"})
public class Player {
```

- Fix any import statements.

- Create the `rummy.jaxb.Link` class.
  - The Link class is used to represent hyperlinks in responses.
  - Create a plain Java class in the `rummy.jaxb` package named Link.
  - Create two public fields, a String href and a String rel. Both should be required XML attributes.
  - Link should be a valid root element and href should come before rel when serializing to XML or JSON.

```
@XmlRootElement
@XmlType(propOrder={"href", "rel"})
public class Link {

    @XmlAttribute(required=true)
    public String href;
    @XmlAttribute(required=true)
    public String rel;

}
```

10. Create the IndianRummyRootResource class.
    - Right-click the IndianRummyWS project and choose New -> Other.
    - Select a category of Java and a file type of Java Class.
    - Click Next.
    - Set the Class Name to IndianRummyRootResource.
    - Set the Package name to rummy.
    - Click Finish.
11. Modify the IndianRummyRootResource class to represent a root resource.
    - Add JAX-RS annotations to the IndianRummyRootResource class. The root resource should be available at the "/" path and produce and consume XML and JSON.

```
@Path("/")
@Produces({MediaType.APPLICATION_XML,
MediaType.APPLICATION_JSON})
@Consumes({MediaType.APPLICATION_XML,
MediaType.APPLICATION_JSON})
```

    - Fix any import statements.
    - Save the class. You should see a REST Resources Configuration dialog box.
      - Select User is responsible for REST resource registration.
      - Deselect Add Jersey library (JAX-RS reference implemention) to project classpath.
      - Click OK.
12. Create the `javax.ws.rs.core.Application` subclass.
    - Right-click the IndianRummyWS project and choose New > Other.
    - Select a category of Java and a file type of Java Class.
    - Click Next.
    - Set the Class Name to RummyApplication.
    - Set the Package name to rummy.
    - Click Finish.
    - Modify the class to extend `javax.ws.rs.core.Application`.

```
public class RummyApplication extends Application
```

    - Annotate the class with the `@ApplicationPath` annotation.

```
@ApplicationPath("resources")
public class RummyApplication extends Application
```

- Override the `getClasses` method to return a Set containing the `IndianRummyRootResource` class.

```
@Override
public Set<Class<?>> getClasses() {
    Set<Class<?>> s = new HashSet<Class<?>>();
    s.add(IndianRummyRootResource.class);
    return s;
}
```

- Fix any imports and save the file.

13. Add `D:\labs\student\exercises\lab10\PlayersResource.java` to the project.
    - Using the Favorites tab in NetBeans copy the `D:\labs\student\exercises\lab10\PlayersResource.java` file.
    - Using the Projects tab in NetBeans paste the `PlayersResource.java` file to the rummy package of the IndianRummyWS project.
    - View the `PlayersResource.java` file.
    - Notice there is no @Path annotation. The path for this resource will be determined by the change you make in the `IndianRummyRootResource` class in the next step.
        - Locate each resource method. You should be able to find four methods with annotations such as @GET, @PUT, and @DELETE. Read through each method to understand its behavior. Pay special attention to the @PUT method that is idempotent and can be used to create or update a player.
14. Update the root resource to provide clients access to the /players sub-resource.
    - Open the IndianRummyRootResource class.
    - Create a `getPlayerResource()` sub-resource locator method that is invoked for requests to the `/players` path.

```
@Path("/players")
public PlayersResource getPlayerResource() {
    return new PlayersResource();
}
```

    - Create a getLinks resource method that is invoked for GET requests to the "/" path.
        - This application will contain two sub-resources: `"/players"` and `"/games"`. Clients should be able to start at the base URL of the service and discover all the sub-resources in the application. For now, only `"/players"` will exist.

```
@GET
public List<Link> getLinks(@Context UriInfo uriInfo) {
    List<Link> links = new ArrayList<>();
    UriBuilder uriBuilder = uriInfo.getAbsolutePathBuilder();

    Link playerLink = new Link();
    UriBuilder playerUri = uriBuilder.clone().path("/players");
    playerLink.href = playerUri.build().toString();
    playerLink.rel = "players";
```

```
        links.add(playerLink);

        return links;
    }
```

- Save all your changes.

15. Test the IndianRummyWS application.
    - Deploy the IndianRummyWS.
    - Open up the FireFox web browser and launch the RESTClient extension.
    - Send a GET request to the http://localhost:7001/IndianRummyWS/resources/ URL.
    - The Response Headers should include a status code of 200 OK and the response body should contain:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<links>
  <link
href="http://localhost:7001/IndianRummyWS/resources/players"
rel="players"/>
</links>
```

    - Copy the players URL from the above response and submit a GET request to that resource using the RESTClient extension. From the /players resource you should receive a response body resembling:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<players>
</players>
```

    - To discover the URL used to create a new player, submit another request to the http://localhost:7001/IndianRummyWS/resources/players URL but use the OPTIONS HTTP method. The response body should contain the WSDL for the IndianRummyWS RESTful application.
    - Create a new player named matt.
        - Enter a URL of http://localhost:7001/IndianRummyWS/resources/players/matt
        - Change the method to PUT.
        - Add a header of Content-Type: `application/xml`
        - Enter the following body and click SEND.

          WARNING! The request body must not have any whitespace before the XML preamble.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<player>
  <userName>matt</userName>
  <password>welcome2</password>
</player>
```

    - View the response headers. You should see a 201 Created status code. Change the password value in the request body to `welcome1` and click SEND again. View the response headers. You should see a 200 OK status because the resource was updated and not created.

- Change the HTTP method to GET and retrieve the matt resource. The response body should resemble:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<player xmlns:ns2="urn:dukesdecks">
  <userName>matt</userName>
  <stats>
    <wins>0</wins>
    <losses>0</losses>
  </stats>
<href>http://localhost:7001/IndianRummyWS/resources/players/matt
</href>
</player>
```

- GET the players resource again. You should see a response like:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<players>
  <player xmlns:ns2="urn:dukesdecks">
    <userName>matt</userName>
<href>http://localhost:7001/IndianRummyWS/resources/players/matt
</href>
  </player>
</players>
```

- Using the http://localhost:7001/IndianRummyWS/resources/players/matt URL, delete the matt resource. When deleting a player resource the body of the request is not used. You should receive a 200 OK response. You can also verify that the resource has been deleted by requesting the players resource again.

16. Note that if an operation fails for some reason you may not see any errors in the RESTClient extension. Remember to look at the Oracle WebLogic Server output tab in NetBeans if things do not appear to be executing correctly. A later lesson will focus on exception and error handling.

Practice 10-3: Creating the Indian Rummy Game Creation REST Resources

Overview

In this practice, you create a game of Indian Rummy. More specifically you create the RummyGameCollectionResource and RummyGameResource. The full mechanics of playing the game will not be implemented in this practice but you should be able to create and deal the initial hand of Indian Rummy.

A "game" of Indian Rummy is stored in two places. Using the CardGameService SOAP web service the card piles used by a game are stored in memory. The additional details such as the order of players at the table and which player's turn it is are stored as serialized XML data into a group description using the user-management-service SOAP web service. Each game of Indian Rummy will be stored as a group named "game-GENERICGAMEUUID" where GENERICGAMEUUID is the identifier returned by the CardGameService when creating a generic card game. The "game-" prefix is used to enable rapid identification and deletion of groups that correspond to rummy games.

Unlike the player REST resources, for the game resources you will use multiple class files to create resource representations.

Assumptions

You have completed Practice 10-2: Creating the Indian Rummy Web Service Project.

Tasks

1. Create two plain Java classes.
   - `rummy.RummyGameCollectionResource`
   - `rummy.RummyGameResource`
2. Create the additional JAXB class used by the application.
   - JAXB classes will be used to create both XML and JSON representation of resources.
   - Create the `rummy.jaxb.RummyGame` class.
     - Create a plain Java class in the `rummy.jaxb` package named RummyGame.
     - Add four JAXB-annotated fields.

```
@XmlElement(name = "player")
private List<Player> players = new ArrayList<>();
@XmlElement(name = "whos-turn")
private String currentTurnPlayer;
@XmlElement(name = "discard-pile-top-card")
private CardType topDiscardCard;
@XmlElement(name = "href")
private String href;
```

     - Fix the import statements.
     - Add getter and setter methods for all fields. Right-click the source window and choose Insert Code > Getter and Setter. Select all fields and click Generate.
     - Add an addPlayer method.

```
public void addPlayer(Player player) {
```

```
    players.add(player);
}
```

- Annotate the class with JAXB annotations.

```
@XmlRootElement(name = "rummy-game")
@XmlType(propOrder = {"players", "currentTurnPlayer",
"topDiscardCard", "href"})
@XmlAccessorType(XmlAccessType.NONE)
public class RummyGame {
```

- Fix the import statements.

3. Update the IndianRummyRootResource class to use a sub-resource locator method to return a RummGameCollectionResource instance for the `"/games"` path.

```
@Path("/games")
public RummyGameCollectionResource getRummyGames() {
    return new RummyGameCollectionResource();
}
```

- Also update the getLinks method to include a link to the games resource.

```
Link gamesLink = new Link();
UriBuilder gamesUri = uriBuilder.clone().path("/games");
gamesLink.href = gamesUri.build().toString();
gamesLink.rel = "games";
links.add(gamesLink);
```

- Save your changes.

4. Add web service client methods to the `RummyGameCollectionResource` class.
   - Right-click in the source window of the `RummyGameCollectionResource` class and choose Insert Code > Call Web Service Operation. Repeat this step as needed to add the following operations:
   - CardGameService > CardGameService > CardGameServicePort
     - create-all-supply-pile
     - createCardGame
     - createPile
     - createPiles
     - removeCardGame
     - showTopCard
     - shuffleAll
   - user-management-service > user-management-service > user-management-port
     - check-group-membership
     - create-group
     - delete-group
     - is-group-present
     - is-user-present
     - join-group
     - list-group-members

- list-groups-by-pattern
- set-group-description

5. Add a private dealGame method to the `RummyGameCollectionResource` class.
   - The dealGame method should perform the following actions:
     - Shuffle all the cards for a given gameId.
     - Get all the members of the group whose name is the gameId.
     - Create a list of strings where each element has a value of "player-USERNAME" where USERNAME is a username.
     - Distribute 13 cards to each player using the "player-USERNAME" piles in a round-robin fashion. The "player-" prefix is used to avoid potential naming conflicts (users named "discard" or "stock").
     - Create a discard pile for the game containing one card.
     - Create a stock pile for the game containing all the remaining cards.

```
private void dealGame(String gameId) {
    shuffleAll(gameId);
    List<String> userNames = listGroupMembers("game-" + gameId);
    for (int i = 0; i < userNames.size(); i++) {
        userNames.set(i, "player-" + userNames.get(i));
    }
    createPiles(gameId, 13, DistributionMethod.ROUND_ROBIN,
userNames);
    createPile(gameId, 1, "discard");
    createAllSupplyPile(gameId, "stock");
}
```

6. Add two private methods, `giveEachPlayerOneCard` and `getPlayerWithHighestCard`, which can be used to figure out which player gets to go first.
   - Add the `giveEachPlayerOneCard` method to the `RummyGameCollectionResource` class. The method should shuffle all the cards in a given game and create a "player-USERNAME" pile for each player containing one card.

```
private void giveEachPlayerOneCard(String gameId) {
    shuffleAll(gameId);
    List<String> userNames = listGroupMembers("game-" + gameId);
    for (String userName : userNames) {
        String playerPile = "player-" + userName;
        createPile(gameId, 1, playerPile);
    }
}
```

   - Add the `getPlayerWithHighestCard` method to the `RummyGameCollectionResource` class. The method should return which player for a given game has the highest valued card at the top of their "player-USERNAME" pile. Return null if there is a tie among the players with the highest rank. User the `util.CardUtils.compareCards` method to compare the rank of two cards.

```
private String getPlayerWithHighestCard(String gameId) {
    List<String> userNames = listGroupMembers("game-" + gameId);
    String topPlayer = "";
    String highRank = "2";
    boolean tie = false;
    for (String userName : userNames) {
        String playerPile = "player-" + userName;
        CardType playerCard = showTopCard(gameId, playerPile);
        int result = CardUtils.compareCards(highRank,
playerCard.getRank());
        if (result == 0) {
            tie = true;
            topPlayer = userName;
        } else if (result == -1) {
            highRank = playerCard.getRank();
            tie = false;
            topPlayer = userName;
        }
    }
    if (tie) {
        return null;
    } else {
        return topPlayer;
    }
}
```

7. Add a logger instance using a logger name of "rummy" to the `RummyGameCollectionResource` class.

```
private static final Logger logger = Logger.getLogger("rummy");
```

8. Create the `createGame` method in the `RummyGameCollectionResource` class that responds to POST requests. Unlike the player resource that used a PUT method to create a new resource, the game collection uses a POST request because the ID of the game is not known until after the game resource is created.
   - The method will have a Response object as the return type, a UriInfo injected parameter used to build the response, and a RummyGame rummyGame parameter that is built using the request body. The incoming RummyGame contains the list of players that wish to join a game.

```
@POST
public Response createGame(@Context UriInfo uriInfo, RummyGame
rummyGame) {
    ResponseBuilder rb;
```

   - All players and groups used should be verified to be unique, exist, and belong to the players group; if they do not then a `BAD_REQUEST` response should be returned.

```
Set<String> s = new HashSet<>();
```

```
for (Player player : rummyGame.getPlayers()) {
    String userName = player.getUserName();
    if (!s.add(userName)
            || !isGroupPresent("players")
            || !isUserPresent(userName)
            || !checkGroupMembership(userName, "players")) {
        logger.log(Level.WARNING, "Not player: {0}", userName);
        rb = Response.status(Response.Status.BAD_REQUEST);
        return rb.build();
    }
}
```

- Using the `CardGameService` web service a new game with the correct number of card decks should be created. If an incompatible number of players are joining a game a `BAD_REQUEST` response should be returned.

```
String gameId;
int players = rummyGame.getPlayers().size();
if (players >= 2 && players <= 3) {
    gameId = createCardGame(1, 2);
} else if (players >= 4 && players <= 6) {
    gameId = createCardGame(2, 2);
} else if (players >= 7 && players <= 10) {
    gameId = createCardGame(3, 2);
} else {
    rb = Response.status(Response.Status.BAD_REQUEST);
    return rb.build();
}
```

- Create a RummyGame instance and using the gameId of the game you just created, create a group using the user-management-service web service. The group should be named "game-GAMEID" where GAMEID is replaced with the actual game ID value. All players in the game should be added to the group.

```
RummyGame newGame = new RummyGame();
createGroup("game-" + gameId, "");
for (Player player : rummyGame.getPlayers()) {
    String userName = player.getUserName();
    joinGroup(userName, "game-" + gameId);
    newGame.addPlayer(player);
}
```

- Using the private `giveEachPlayerOneCard` and `getPlayerWithHighestCard` methods, determine which player goes first and save that information in the RummyGame instance.

```
String firstPlayer = null;
while (firstPlayer == null) {
    giveEachPlayerOneCard(gameId);
```

```
    firstPlayer = getPlayerWithHighestCard(gameId);
}
newGame.setCurrentTurnPlayer(firstPlayer);
```

- Deal the game.

```
dealGame(gameId);
```

- Using a `JAXBContext` and a `ByteArrayOutputStream`, convert the RummyGame instance into an XML string and save it as the description for the "game-GAMEID" group. If this fails for some reason log a message and attempt to delete the "game-GAMEID" group.

```
try {
    JAXBContext jaxbContext =
JAXBContext.newInstance(RummyGame.class, CardType.class);
    Marshaller m = jaxbContext.createMarshaller();
    ByteArrayOutputStream bout = new ByteArrayOutputStream();
    Writer writer = new OutputStreamWriter(bout);
    m.marshal(newGame, writer);
    String gameStr = new String(bout.toByteArray());
    setGroupDescription("game-" + gameId, gameStr);
} catch (JAXBException ex) {
    logger.log(Level.SEVERE, "Failed to generated game XML",
ex);
    deleteGroup("game-" + gameId);
    rb = Response.status(Response.Status.INTERNAL_SERVER_ERROR);
    return rb.build();
}
```

- Using the uriInfo, return a CREATED response to the new game.

```
UriBuilder uriBuilder = uriInfo.getAbsolutePathBuilder();
uriBuilder.path(gameId);
rb = Response.created(uriBuilder.build(newGame));
return rb.build();
```

9. Create the `getGames` method in the `RummyGameCollectionResource` class that responds to GET requests.
   - This method calls `listGroupsByPattern("games-*")` to obtain a list of rummy group names that are used to construct a list of RummyGame instances with only a game ID and href set.

```
@GET
public List<RummyGame> getGames(@Context UriInfo uriInfo) {
    List<RummyGame> list = new ArrayList<>();
    UriBuilder uriBuilder = uriInfo.getAbsolutePathBuilder();
    for (String groupName : listGroupsByPattern("game-*")) {
        RummyGame rg = new RummyGame();
        String gameId = groupName.substring(5);
```

```
rg.setHref(uriBuilder.clone().path(gameId).build().toString());
        list.add(rg);
    }
    return list;
}
```

10. Create the `deleteGames` method in the `RummyGameCollectionResource` class that responds to DELETE requests.
    - This method calls `listGroupsByPattern("games-*")` to obtain a list of rummy group names and then deletes them from the CardGameService web service and removes each corresponding group from the user-management-service web service.

```
@DELETE
public void deleteGames() {
    for (String groupName : listGroupsByPattern("game-*")) {
        String gameId = groupName.substring(5);
        removeCardGame(gameId);
        deleteGroup(groupName);
    }
}
```

11. Add a `getGameResource` sub-resource locator method to the `RummyGameCollectionResource` class.

```
@Path("{id}")
public RummyGameResource getGameResource() {
    return new RummyGameResource();
}
```

12. Open the `RummyGameResource.java` file. This resource will respond to requests to URLs similar to http://localhost:7001/IndianRummyWS/resources/games/0d2756c5-45bd-4a98-b5c3-1ae20a7032fa.
13. Add web service client methods to the `RummyGameResource` class.
    - Right-click in the source window of the `RummyGameResource` class and choose Insert Code > Call Web Service Operation. Repeat this step as needed to add the following operations:
    - CardGameService > CardGameService > CardGameServicePort
      - showTopCard
      - showPile
      - removeCardGame
    - user-management-service > user-management-service > user-management-port
      - delete-group
      - get-group-description
      - is-group-present
14. Create the getGame resource method that responds to HTTP GET requests.
    - The getGame method uses a Response object to return different HTTP status codes to clients.

```
@GET
public Response getGame(@Context UriInfo uriInfo,
@PathParam("id") String gameId) {
    ResponseBuilder rb;
```

- Build the RummyGame using the XML stored in the group description.

```
RummyGame game;
try {
    JAXBContext jaxbContext =
JAXBContext.newInstance(RummyGame.class);
    Unmarshaller u = jaxbContext.createUnmarshaller();
    String groupDesc = getGroupDescription("game-" + gameId);
    ByteArrayInputStream in = new
ByteArrayInputStream(groupDesc.getBytes());
    game = (RummyGame) u.unmarshal(in);
} catch (JAXBException ex) {
    logger.log(Level.SEVERE, "Failed to read game XML", ex);
    rb = Response.status(Response.Status.INTERNAL_SERVER_ERROR);
    return rb.build();
}
```

- Get the top card from the discard pile and add it to the RummyGame instance.

```
    CardType topDiscardCard = showTopCard(gameId, "discard");
    game.setTopDiscardCard(topDiscardCard);
```

- Loop through the list of players for the game and add a Hand object with only an href value to the game for each player.

```
for (Player player : game.getPlayers()) {
    Hand hand = new Hand();
    player.setHand(hand);
    UriBuilder uriBuilder = uriInfo.getAbsolutePathBuilder();
    URI uri = uriBuilder.path("/hands/" +
player.getUserName()).build();
    hand.setHref(uri.toString());
}
```

- Return the game in an HTTP 200 OK response.

```
game.setHref(uriInfo.getAbsolutePath().toString());
return Response.ok(game).build();
```

15. Create the `deleteGame` resource method that responds to HTTP DELETE requests.
    - The `deleteGame` method uses a Response object to return either a 200 OK status code if the game was deleted or a 404 NOT FOUND status code if no such game exists.

```
@DELETE
public Response deleteGame(@PathParam("id") String gameId) {
    ResponseBuilder rb;
```

```
    if (!isGroupPresent("game-" + gameId)) {
        rb = Response.status(Response.Status.NOT_FOUND);
        return rb.build();
    } else {
        removeCardGame(gameId);
        deleteGroup("game-" + gameId);
        return Response.ok().build();
    }
}
```

16. Create the `getHand` sub-resource method that responds to HTTP GET requests at the `"/hands/{user}"` path.
    - The `getHand` method uses a Response object to return either a 200 OK status code or a 404 NOT FOUND status code.

```
@GET
@Path("/hands/{user}")
public Response getHand(@Context UriInfo uriInfo,
@PathParam("id") String gameId, @PathParam("user") String
userName) {
    ResponseBuilder rb;
```

    - If the group for this game does not exist return a 404 NOT FOUND status code.

```
if (!isGroupPresent("game-" + gameId)) {
    rb = Response.status(Response.Status.NOT_FOUND);
    return rb.build();
}
```

    - Attempt to obtain the hand of the player identified by the "user" path parameter. A player's hand is just a card pile named "player-USERNAME". If the result is null or empty, return a 404 NOT FOUND result.

```
List<CardType> cards = showPile(gameId, "player-" + userName);
if (cards == null || cards.isEmpty()) {
    rb = Response.status(Response.Status.NOT_FOUND);
    return rb.build();
}
```

    - Add the cards to a new Hand object and add the hands URL to itself.

```
Hand hand = new Hand();
hand.setCards(cards);
UriBuilder uriBuilder = uriInfo.getAbsolutePathBuilder();
URI uri = uriBuilder.path("/hands/" + userName).build();
hand.setHref(uri.toString());
```

    - Return the hand using a 200 OK status code.

```
return Response.ok(hand).build();
```

    - Add any required import statements.
    - Save all your changes.

17. Test the IndianRummyWS application.
    - Deploy the IndianRummyWS.
    - Open up the FireFox web browser and launch the RESTClient extension.
    - Send a GET request to the http://localhost:7001/IndianRummyWS/resources/ URL.
    - The Response Headers should include a status code of 200 OK and the response body should contain two links now:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<links>
  <link
href="http://localhost:7001/IndianRummyWS/resources/players"
rel="players"/>
  <link
href="http://localhost:7001/IndianRummyWS/resources/games"
rel="games"/>
</links>
```

    - Create a new player named matt.
        - Enter a URL of http://localhost:7001/IndianRummyWS/resources/players/matt
        - Change the method to PUT.
        - Add a header of Content-Type: `application/xml`
        - Enter the following body and click SEND.

          WARNING! The request body must not have any whitespace before the XML preamble.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<player>
  <userName>matt</userName>
  <password>welcome1</password>
</player>
```

    - Create a new player named tom.
        - Enter a URL of http://localhost:7001/IndianRummyWS/resources/players/tom
        - Change the method to PUT.
        - There should be a header of Content-Type: `application/xml`.
        - Enter the following body and click SEND.

          WARNING! The request body must not have any whitespace before the XML preamble.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<player>
  <userName>tom</userName>
  <password>welcome1</password>
</player>
```

    - Change the HTTP method to GET and retrieve the matt resource. The response body should resemble:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<player xmlns:ns2="urn:dukesdecks">
```

```
  <userName>matt</userName>
  <stats>
    <wins>0</wins>
    <losses>0</losses>
  </stats>
<href>http://localhost:7001/IndianRummyWS/resources/players/matt
</href>
</player>
```

- GET the players resource again. You should see a response like:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<players>
  <player xmlns:ns2="urn:dukesdecks">
    <userName>matt</userName>
<href>http://localhost:7001/IndianRummyWS/resources/players/matt
</href>
  </player>
</players>
```

- Using the http://localhost:7001/IndianRummyWS/resources/games URL, obtain the current list of games (there should not be any).

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<rummyGames>
</rummyGames>
```

- Using the http://localhost:7001/IndianRummyWS/resources/games URL, create a new game using the POST method with a body indicating that matt and tom wish to play a game.
  - There should be a header of Content-Type: `application/xml`

    WARNING! The request body must not have any whitespace before the XML preamble.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<rummy-game>
  <player>
    <userName>tom</userName>
  </player>
  <player>
    <userName>matt</userName>
  </player>
</rummy-game>
```

- View the response headers after creating a game. You should see a 201 CREATED status code along with the URL of the created resource. You can also discover the URL by performing a GET request to the games resource. The URL should resemble:

```
http://localhost:7001/IndianRummyWS/resources/games/a4ca65b2-
572f-4647-974e-33dc3e2838f1
```

- Using the URL of the game that was just created, perform a GET request on that particular game resource. You should receive a response resembling:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<rummy-game xmlns:ns2="urn:dukesdecks">
  <player>
    <userName>tom</userName>
    <hand>
<href>http://localhost:7001/IndianRummyWS/resources/games/a4ca65
b2-572f-4647-974e-33dc3e2838f1/hands/tom</href>
    </hand>
  </player>
  <player>
    <userName>matt</userName>
    <hand>
<href>http://localhost:7001/IndianRummyWS/resources/games/a4ca65
b2-572f-4647-974e-33dc3e2838f1/hands/matt</href>
    </hand>
  </player>
  <whos-turn>matt</whos-turn>
  <discard-pile-top-card>
    <ns2:rank>9</ns2:rank>
    <ns2:suit>HEARTS</ns2:suit>
  </discard-pile-top-card>
<href>http://localhost:7001/IndianRummyWS/resources/games/a4ca65
b2-572f-4647-974e-33dc3e2838f1</href>
</rummy-game>
```

- Using the URLs in the preceding response, perform a GET request to the hand resources for tom and matt.
- Delete the rummy game instance by making a DELETE request to the game URL that is similar to http://localhost:7001/IndianRummyWS/resources/games/a4ca65b2-572f-4647-974e-33dc3e2838f1. You should receive a 200 OK response header.
- Delete the tom and matt users.

Practice 10-4: Using JSON as a Data Interchange Format

Overview

In this practice, you use JSON instead of XML when communicating with your RESTful web service.

The Jersey implementation of JAX-RS supports the JSON interchange format using a library named Jackson. As long as the Jackson-related JARs are on your CLASSPATH, you can use JSON. These JARs should already be present for the classroom environment.

Because these JARS are already present and all your resource classes indicated that they supported JSON:

```
@Produces({MediaType.APPLICATION_XML,
MediaType.APPLICATION_JSON})
@Consumes({MediaType.APPLICATION_XML,
MediaType.APPLICATION_JSON})
```

You can use JSON already with your deployed project.

If desired, you may also use the cURL command-line utility to perform this practice.

Assumptions

You have completed Practice 10-3: Creating the Indian Rummy Game Creation REST Resources.

WebLogic Server is running and the IndianRummyWS project is deployed.

Tasks

1. Open up the FireFox web browsers and launch the RESTClient extension.
2. Remove any headers that might already be present and add two headers to use for the remainder of the tasks:
   - Content-Type: `application/json`
   - Accept: `application/json`
3. Send a GET request to the http://localhost:7001/IndianRummyWS/resources/ URL.
   - The Response Headers should include a status code of 200 OK and the response body should contain two links now:

```
{
   "link":
   [
       {
           "@href":
"http://localhost:7001/IndianRummyWS/resources/players",
           "@rel": "players"
       },
       {
           "@href":
"http://localhost:7001/IndianRummyWS/resources/games",
           "@rel": "games"
       }
```

```
    ]
}
```

- Notice the `@` before the `href` and `rel` names. This is because they are annotated with `@XmlAttribute` instead of `@XmlElement`.

4. Create a new player named matt.
   - Enter a URL of http://localhost:7001/IndianRummyWS/resources/players/matt
   - Change the method to PUT.
   - Enter the following body and click SEND.

```
{
    "userName" : "matt",
    "password" : "welcome1"
}
```

   - View the response headers and you should see a 201 Created status code. Changing the password value in the request body and sending the same request again would result in a 200 OK status because the resource would be updated and not created.

5. GET the players listing.
   - Enter a URL of http://localhost:7001/IndianRummyWS/resources/players.
   - Change the method to GET.
   - Enter the following body and click SEND.

```
{
   "player":
   {
       "userName": "matt",
       "href":
"http://localhost:7001/IndianRummyWS/resources/players/matt"
   }
}
```

6. Create a new player named tom.
   - Enter a URL of http://localhost:7001/IndianRummyWS/resources/players/tom.
   - Change the method to PUT.
   - Enter the following body and click SEND.

```
{
    "userName" : "tom",
    "password" : "welcome1"
}
```

7. GET the players listing again.
   - Enter a URL of http://localhost:7001/IndianRummyWS/resources/players.
   - Change the method to GET.
   - Enter the following body and click SEND.

```
{
   "player":
   [
```

```
        {
            "userName": "tom",
            "href":
"http://localhost:7001/IndianRummyWS/resources/players/tom"
        },
        {
            "userName": "matt",
            "href":
"http://localhost:7001/IndianRummyWS/resources/players/matt"
        }
    ]
}
```

- Notice that the value of "player" is an array (square brackets []). When a single element collection or array was returned as was in task 5, there was no surrounding array.

8. Change the HTTP method to GET and retrieve the matt resource. The response body should resemble:

```
{
    "userName": "matt",
    "stats":
    {
        "wins": "0",
        "losses": "0"
    },
    "href":
"http://localhost:7001/IndianRummyWS/resources/players/matt"
}
```

9. Using the http://localhost:7001/IndianRummyWS/resources/games URL, obtain the current list of games (there should not be any). If you view the Response Body (Raw) tab you should see that the body was null (not present).
10. Using the http://localhost:7001/IndianRummyWS/resources/games URL, create a new game using the POST method with a body indicating that matt and tom wish to play a game.

```
{
    "player" : {
        "userName" : "tom"
    },
    "player" : {
        "userName" : "matt"
    }
}
```

- View the response headers after creating a game. You should see a 201 CREATED status code along with the URL of the created resource. You can also discover the URL by performing a GET request to the games resource. The URL should resemble:

```
http://localhost:7001/IndianRummyWS/resources/games/8ac9317f-
2e0c-495b-9a4b-9bfcfd9a3fe9
```

11. Using the URL of the game that was just created, perform a GET request on that particular game resource. You should receive a response resembling:

```
{
   "player":
   [
       {
           "userName": "tom",
           "hand":
           {
               "href":
"http://localhost:7001/IndianRummyWS/resources/games/8ac9317f-
2e0c-495b-9a4b-9bfcfd9a3fe9/hands/tom"
           }
       },
       {
           "userName": "matt",
           "hand":
           {
               "href":
"http://localhost:7001/IndianRummyWS/resources/games/8ac9317f-
2e0c-495b-9a4b-9bfcfd9a3fe9/hands/matt"
           }
       }
   ],
   "whos-turn": "tom",
   "discard-pile-top-card":
   {
       "rank": "5",
       "suit": "CLUBS"
   },
   "href":
"http://localhost:7001/IndianRummyWS/resources/games/8ac9317f-
2e0c-495b-9a4b-9bfcfd9a3fe9"
}
```

12. Using the URLs in the preceding response, perform a GET request to the hand resources for tom and matt.
13. The default formatting of JSON data does not always match client expectations. For example, if your JavaScript code expects an array but doesn't receive one because there was only a single element, it takes more code to handle the different types of objects you must process. The following steps show you how to modify the JSON output to use natural formatting instead of the default mapped formatting.
14. Create a custom ContextResolver<JAXBContext> to modify the formatting of JSON produced by the application.

- Create a Java class named JAXBContextResolver in the rummy package.
- The class should be annotated with the @Provider annotation and implement ContextResolver<JAXBContext>

```
@Provider
public class JAXBContextResolver implements
ContextResolver<JAXBContext> {
```

- Create two fields, a JAXBContext that will be initialized in the constructor and a Class[] named types that contains all the JAXB annotated classes used by the project.

```
private final JAXBContext context;
private Class[] types = {Hand.class,
    Link.class,
    Player.class,
    PlayerStats.class,
    RummyGame.class,
    CardType.class};
```

- Add a constructor that builds a JSONConfiguration and passes it and the types array to the constructor of a new JSONJAXBContext. Save the JSCONJAXBContext instance in the context field.

```
public JAXBContextResolver() throws Exception {
    JSONConfiguration jsonConfig = JSONConfiguration.natural()
            .rootUnwrapping(false)
            .humanReadableFormatting(true)
            .build();
    context = new JSONJAXBContext(jsonConfig, types);
}
```

- Override the getContext method to return the value stored in the context field.

```
@Override
public JAXBContext getContext(Class<?> objectType) {
    return context;
}
```

- Add any needed imports.

15. Modify the RummyApplication class to produce a singleton instance of the new JAXBContextResolver class.
    - Open the RummyApplication.java file.
    - Create a getSingletons method that produces a Set that contains an instance of the JAXBContextResolver.

```
@Override
public Set<Object> getSingletons() {
    HashSet<Object> set = new HashSet<>(1);
    try {
        JAXBContextResolver myJAXBContextResolver = new
JAXBContextResolver();
        set.add(myJAXBContextResolver);
```

```
    } catch (Exception ex) {
        ex.printStackTrace();
    }
    return set;
}
```

- Add any needed imports.
- Save all your changes and redeploy the application.

16. Try requesting a list of games in JSON format using the RESTClient extension. The Response Body (Raw) tab should contain formatted output now, resembling:

```
{"rummy-game":[ {
  "href" :
"http://localhost:7001/IndianRummyWS/resources/games/8ac9317f-
2e0c-495b-9a4b-9bfcfd9a3fe9"
} ]}
```

- Even though there is only a single game, you now see a one-element array as the value unlike before when you saw a different behavior in tasks 5 and 7.

17. Perform a GET request of the root resource, http://localhost:7001/IndianRummyWS/resources/.
    - You should receive a response that resembles:

```
{"rummy.jaxb.Link":[ {
  "href" :
"http://localhost:7001/IndianRummyWS/resources/players",
  "rel" : "players"
}, {
  "href" :
"http://localhost:7001/IndianRummyWS/resources/games",
  "rel" : "games"
} ]}
```

    - In a previous request, task 3, of the data, the href and rel values were preceded by an @.
    - Also notice the value of rummy.jaxb.Link as the key for the root object in the response. Open the Link.java file and add a name attribute to the @XmlRootElement annotation.

```
@XmlRootElement(name="link")
```

    - Save your changes and GET the root resource again. You should now see:

```
{"link":[ {
  "href" :
"http://localhost:7001/IndianRummyWS/resources/players",
  "rel" : "players"
}, {
  "href" :
"http://localhost:7001/IndianRummyWS/resources/games",
  "rel" : "games"
} ]}
```

18. Modify the JAXBContextResolver to enable root unwrapping.

- Open the JAXBContextResolver class.
- Modify the line that says

```
.rootUnwrapping(false)
```

  to say

```
.rootUnwrapping(true)
```

- Save your changes and GET the root resource again. You should now see:

```
[ {
  "href" :
"http://localhost:7001/IndianRummyWS/resources/players",
  "rel" : "players"
}, {
  "href" :
"http://localhost:7001/IndianRummyWS/resources/games",
  "rel" : "games"
} ]
```

- The "link" root JSON element is now stripped out.

(Optional) Practice 10-5: Completing the Indian Rummy Logic

Overview

In this practice, you finish the Indian Rummy resources to enable playing single-round games.

Assumptions

You are really fast at these practices and need a big challenge (no solution is provided).

Tasks

1. Design the additional resources and determine which operations would be required to support playing a round of Indian Rummy. Discuss your design with other students and your instructor. Does your design take hypermedia into consideration? Are your resources (URLs) nouns or verbs?
2. Make a copy of the IndianRummyWS project. You can continue to add features to the copy when time permits throughout the rest of the week.
3. Complete the IndianRummyWS using either your design or the one outlined below.
   - All operations should ensure that the resources in use are valid, throwing 404 for a non-existent game, and so on.
   - Create the stock and discard resources.
     - games/UUID/stock
     - games/UUID/discard
   - Performing a GET to either of these resources will take a card from the implied card pile and add it to the hand of the player whose turn it is. If that player already has 14 cards in hand, do nothing.
   - Performing a PUT to games/UUID/discard will take the card specified in the request body from the player whose turn it is and put it into the discard pile. This happens only if they have 14 cards and a card in hand that matches the card to be discarded. Otherwise, a base request response should be returned.
   - Performing a PUT of 13 cards to games/UUID will attempt to rummy. The 13 cards must match the cards in the hand of the player whose turn it is. That player must also have 14 cards in hand at the time. The 13 cards should be validated (life1, life2, etc). The win/loss totals of all players should be updated and the game should be destroyed.
   - Add filtering parameters to the games and players resources. Return 10 items per page and use an `offset` query param to select the starting item on a page. Links with `next` and `previous` rel attributes should be returned.
   - Completely implement HATEOAS. Some parts of the application cannot be discovered by starting at the root resource. How would a client know how to create a player? You need a REST equivalent of HTML forms. The /resources/players resource should contain a link to the "form" for creating a new player using a Link header with a "new" relationship. The "form" (/resources/new-player) would just be a sample of the request body that needs to be submitted to /resources/players/{id}. When obtaining the form from /resources/new-player a Link header with a relationship of "submit" should be returned that tells the client where to submit the form. Repeat for game creation.
   - Support other game variations such as supporting multiple rounds per game (see RummyRules project). Some variations require an extra game state that could be stored as part of XML data stored in the game's group description.

Practices for Lesson 11: Web Service Error Handling

Chapter 11

Practices for Lesson 11: Web Service Error Handling

Practices Overview

In these practices, you will configure a JAX-WS and a JAX-RS web service to handle errors, namely to convert exceptions to the appropriate SOAP or HTTP response.

Practice 11-1: JAX-WS Basic Error Handling

Overview

In this practice you modify the GenericCardGameWS bottom-up (code first) web service to produce SOAP faults. You will then have to modify its client, the IndiaRummyWS, to handle possible faults.

Assumptions

You have completed Practice 10-4: Using JSON as a Data Interchange Format.

Tasks

1. Open the GenericCardGameWS project from practice 8-1 if it is not already open in NetBeans.
2. Start WebLogic Server if it is not already started.
3. Test the CardGameService.
   - Expand the Web Services node for the project, right-click CardGameService, and choose Test Web Service.
   - You can also open a web browser and go to http://localhost:7001/wls_utc/begin.do?wsdlUrl=http://localhost:7001/GenericCardGameWS/CardGameService?wsdl.
   - Create a card game with three decks and two jokers per deck.
   - Copy the complete Service Request text to the clipboard.
4. Submit a malformed request using the RESTClient FireFox extension.
   - Launch the RESTClient FireFox extension.
   - Add a header, Content-Type: text/xml
   - Set the URL to http://localhost:7001/GenericCardGameWS/CardGameService.
   - Change the HTTP Method to POST.
   - Complete the Body by pasting the clipboard content from the previous step.

```
<env:Envelope
xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
  <env:Header />
    <env:Body>
    <createCardGame xmlns="http://games/">
      <deck-count xmlns="">3</deck-count>
      <jokers-per-deck xmlns="">2</jokers-per-deck>
    </createCardGame>
  </env:Body>
</env:Envelope>
```

   - Send the request to ensure that everything is valid. You should receive a response with a 200 OK status code and a response body resembling:

```
<?xml version='1.0' encoding='UTF-8'?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
```

```
    <ns2:createCardGameResponse xmlns:ns3="urn:dukesdecks"
xmlns:ns2="http://games/">
      <return>6cec4777-f1b7-4717-b7bc-fe57aaf6ec0e</return>
    </ns2:createCardGameResponse>
  </S:Body>
</S:Envelope>
```

- Modify the request body to have a malformed request.
  - Change the <jokers-per-deck> tags to <jokers>.
- Submit the request. Is a game created? Are there any errors indicated in the response header or response body? Try changing the <jokers> tags to <mike> tags. Does it still work? Try deleting the joker or mike elements and submit.
- Restore the request body to a valid copy.
- Modify the request body to have a malformed request.
  - In the request body change the <createCardGame> tags to <createGame> and send the request.
- Inspect the SOAP fault.
  - In the response headers you should see a 500 Internal Server Error status code.
  - The response body should contain:

```
<?xml version='1.0' encoding='UTF-8'?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <S:Fault xmlns:ns4="http://www.w3.org/2003/05/soap-
envelope">
      <faultcode>S:Client</faultcode>
      <faultstring>Cannot find dispatch method for
{http://games/}createGame</faultstring>
    </S:Fault>
  </S:Body>
</S:Envelope>
```

- JAX-WS can be very forgiving but not every malformed request is accepted.

5. Enable schema validation for document literal web services.
   - In NetBeans add a new Ant Library to enable access to the @SchemaValidation annotation.
   - Open the Tools > Ant Libraries menu item.
   - Click New Library and enter:
     - Library Name: JAX-WS Runtime
     - Library Type: Server Libraries
   - Add the D:\weblogic\wls\modules\glassfish.jaxws.rt_2.0.0.0_2-2-5.jar file to the JAX-WS Runtime library and close the Ant Libraries dialog box.
   - Add the JAX-WS Runtime library to the GenericCardGameWS project.
     - Right-click the Libraries node in the GenericCardGameWS project and select Add Library.
     - Select the JAX-WS Runtime library from the list.

- Open the CardGameService.java file.
- Add the @SchemaValidation annotation to the CardGameService class.
- Add any required imports and save your changes.

  The required import is:

```
import com.sun.xml.ws.developer.SchemaValidation;
```

  Do **NOT** use

```
import com.sun.xml.internal.ws.developer.SchemaValidation;
```

  If you can only find the "internal" package then you have not added the required library JAR correctly.
- Save your changes and deploy the GenericCardGameWS project.
- Repeat the steps from task 4. Changing <jokers-per-deck> to <jokers> should now result in an HTTP status code of 500 Internal Server Error and a response body containing a SOAP fault.

```
<?xml version='1.0' encoding='UTF-8'?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <S:Fault xmlns:ns4="http://www.w3.org/2003/05/soap-
envelope">
      <faultcode>S:Client</faultcode>
      <faultstring>org.xml.sax.SAXParseException; cvc-complex-
type.2.4.a: Invalid content was found starting with element
'jokers'. One of '{jokers-per-deck}' is expected.</faultstring>
    </S:Fault>
  </S:Body>
</S:Envelope>
```

6. Throw a RuntimetimeException from a JAX-WS web service.
   - In NetBeans, open the CardGameService.java file.
   - Locate the createCardGame method.
   - If this is a publicly available web service, what types of invalid input might this method receive even if the SOAP request was well formed? When using a top-down (schema first) approach there might be a restriction that limits the range of numbers allowed. In that case just enabling schema validation might filter an invalid input. In this bottom-up (code first) web service there is no range restriction. How well would this service handle a request to create a card game with 1,000,000 decks? What about -1 decks?
   - At the beginning of the createCardGame method, add code to check the range the method parameters and throw an IllegalArgumentException (a subclass of RuntimeException) when the range is outside what is expected.

```
if(numberOfDecks < 1 || numberOfDecks > 10) {
    throw new IllegalArgumentException("Only 1-10 decks
supported");
}

if(jokerCountPerDeck < 0 || jokerCountPerDeck > 4) {
```

```
    throw new IllegalArgumentException("Only 0-4 jokers per deck
supported");
}
```

- Save your changes and deploy the application.
- Using either the Weblogic Test Client or the RESTClient FireFox extension test the createCardGame operation with an illegal number of decks. You should receive a 500 Internal Server Error response with a body resembling:

```
<?xml version='1.0' encoding='UTF-8'?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <S:Fault xmlns:ns4="http://www.w3.org/2003/05/soap-
envelope">
      <faultcode>S:Server</faultcode>
      <faultstring>Only 1-10 decks supported</faultstring>
    </S:Fault>
  </S:Body>
</S:Envelope>
```

7. Throw a SOAPFaultException from the createCardGame method.
   - Notice that the <faultcode> value is S:Server in the response when throwing a RuntimeException subclass. This indicates to the client that something went wrong on the server when in reality the problem was caused by the client.
   - By using a SOAPFaultException you can gain more control over the faultcode and faultstring that is returned to the client.
   - In NetBeans, open the CardGameService.java file.
   - Locate the createCardGame method.
   - Instead of throwing an IllegalArgumentException, throw a SOAPFaultException.

```
if (numberOfDecks < 1 || numberOfDecks > 10) {
    try {
        QName faultCode = new
QName(SOAPConstants.URI_NS_SOAP_1_1_ENVELOPE, "Client");
        String reasonText = "Only 1-10 decks supported";
        SOAPFault fault =
SOAPFactory.newInstance().createFault(reasonText, faultCode);
        throw new SOAPFaultException(fault);
    } catch (SOAPException ex) {
        throw new IllegalArgumentException("Only 1-10 decks
supported");
    }
}
```

   - Add any required imports. The import of SOAPFaultException should be:

```
import javax.xml.ws.soap.SOAPFaultException;
```

     The import of SOAPConstants should be:

```
import javax.xml.soap.SOAPConstants;
```

- Save your changes and deploy the application.
- Using either the Weblogic Test Client or the RESTClient FireFox extension, test the createCardGame operation with an illegal number of decks. You should receive a 500 Internal Server Error response with a body resembling:

```
<?xml version='1.0' encoding='UTF-8'?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <S:Fault xmlns:ns4="http://www.w3.org/2003/05/soap-
envelope">
      <faultcode>S:Client</faultcode>
      <faultstring>Only 1-10 decks supported</faultstring>
    </S:Fault>
  </S:Body>
</S:Envelope>
```

- A SOAPFaultException can clearly indicate the nature of the failure to your client. In addition to that, you avoid polluting you servers log files with stack traces detailing runtime exceptions.

8. Handle a SOAPFaultException on the client-side.
   - In NetBeans, open the CardGameService.java file.
   - Locate the createCardGame method.
   - Modify the createCardGame method to only support creating one or two deck games.

```
if (numberOfDecks < 1 || numberOfDecks > 2) {
    try {
        QName faultCode = new
QName(SOAPConstants.URI_NS_SOAP_1_1_ENVELOPE, "Client");
        String reasonText = "Only 1-2 decks supported";
        SOAPFault fault =
SOAPFactory.newInstance().createFault(reasonText, faultCode);
        throw new SOAPFaultException(fault);
    } catch (SOAPException ex) {
        throw new IllegalArgumentException("Only 1-2 decks
supported");
    }
}
```

   - Save and deploy the GenericCardGameWS project.
   - As of now all changes to the GenericCardGameWS web service are internal; the WSDL produced by this service remains unchanged.
   - Open the IndianRummyWS project from practice 10-4 if it is not already open, and deploy it.
   - Launch the RESTClient FireFox extension.
   - Create eight new players named duke, larry, peter, james, mike, cindy, joe, and diganta.
     - Enter a URL of http://localhost:7001/IndianRummyWS/resources/players/USERNAME (where USERNAME changes for each new player).

- Change the method to PUT.
- There should be a header of Content-Type: application/xml
- Enter the following body and click SEND.

  WARNING! The request body must not have any whitespace before the XML preamble.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<player>
  <userName>duke</userName>
  <password>welcome1</password>
</player>
```

- Repeat the process until all eight players are created.

- Using the http://localhost:7001/IndianRummyWS/resources/games URL, create a new game using the POST method with a body indicating that duke, larry, peter, james, mike, cindy, joe, and diganta wish to play a game.
  - There should be a header of Content-Type: application/xml

    WARNING! The request body must not have any whitespace before the XML preamble.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<rummy-game>
  <player>
    <userName>duke</userName>
  </player>
  <player>
    <userName>larry</userName>
  </player>
  <player>
    <userName>peter</userName>
  </player>
  <player>
    <userName>james</userName>
  </player>
  <player>
    <userName>mike</userName>
  </player>
  <player>
    <userName>cindy</userName>
  </player>
  <player>
    <userName>joe</userName>
  </player>
  <player>
    <userName>diganta</userName>
  </player>
```

```
</rummy-game>
```

- Upon submitting the request to create a game of Indian Rummy with eight players (which requires three decks) you should get a response of 500 Internal Server Error and a response body containing a stack trace.

- In NetBeans, open the RummyGameCollectionResource.java file in the IndianRummyWS project.
- Find the block of code that functions as a JAX-WS client to the GenericCardGame WS web service.

```
String gameId;
int players = rummyGame.getPlayers().size();
if (players >= 2 && players <= 3) {
    gameId = createCardGame(1, 2);
} else if (players >= 4 && players <= 6) {
    gameId = createCardGame(2, 2);
} else if (players >= 7 && players <= 10) {
    gameId = createCardGame(3, 2);
} else {
    rb = Response.status(Response.Status.BAD_REQUEST);
    return rb.build();
}
```

- The createCardGame method was inserted at the bottom of this class by you when you inserted a call to a web service operation.

```
private static String createCardGame(int deckCount, int
jokersPerDeck) {
    cards.CardGameService_Service service = new
cards.CardGameService_Service();
    cards.CardGameService port =
service.getCardGameServicePort();
    return port.createCardGame(deckCount, jokersPerDeck);
}
```

- It is the call to port.createCardGame(deckCount, jokersPerDeck) that is throwing the SOAPFaultException.
- Find the line that is creating three card decks.

```
gameId = createCardGame(3, 2);
```

- Surround it with a try-catch block that handles a SOAPFaultException.

```
try {
    gameId = createCardGame(3, 2);
} catch (SOAPFaultException ex) {
    rb = Response
            .status(Response.Status.BAD_REQUEST)
            .entity(ex.getFault().getFaultString());
    return rb.build();
}
```

- Add any required imports. The import of SOAPFaultException should be:

```
import javax.xml.ws.soap.SOAPFaultException;
```

- Save your changes and deploy the project.
- Launch the RESTClient FireFox extension.
- Using the http://localhost:7001/IndianRummyWS/resources/games URL, create a new game using the POST method with a body indicating that duke, larry, peter, james, mike, cindy, joe, and diganta wish to play a game.
  - There should be a header of Content-Type: application/xml

    WARNING! The request body must not have any whitespace before the XML preamble.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<rummy-game>
  <player>
    <userName>duke</userName>
  </player>
  <player>
    <userName>larry</userName>
  </player>
  <player>
    <userName>peter</userName>
  </player>
  <player>
    <userName>james</userName>
  </player>
  <player>
    <userName>mike</userName>
  </player>
  <player>
    <userName>cindy</userName>
  </player>
  <player>
    <userName>joe</userName>
  </player>
  <player>
    <userName>diganta</userName>
  </player>
</rummy-game>
```

  - Because the JAX-WS client (RummyGameCollectionResource.java) is catching the SOAPFaultException (as subclass of RuntimeException) the SOAP fault is no longer causing the RummyGameCollectionResource.java code to fail. The REST service is able to catch the exception, set a 400 Bad Request response status and also complete the response body with a single message "Only 1-2 decks supported."

9. Throw a custom application exception as a SOAP fault.

- In NetBeans, switch to the GenericCardGameWS project.
- Create a new games.GameException class that extends Exception.

```
package games;

public class GameException extends Exception {

}
```

- Add four constructors whose argument list matches those from the parent class. The Insert Code tool has a Constructor option that can be used.

```
public GameException() {
}

public GameException(String message) {
    super(message);
}

public GameException(String message, Throwable cause) {
    super(message, cause);
}

public GameException(String message, Throwable cause, boolean
enableSuppression, boolean writableStackTrace) {
    super(message, cause, enableSuppression,
writableStackTrace);
}
```

- Open the CardGameService.java file in the GenericCardGame project.
- Find the createCardGame method.
- Add a throws GameException statement to the method signature.

```
public String createCardGame(@WebParam(name = "deck-count") int
numberOfDecks,
        @WebParam(name = "jokers-per-deck") int
jokerCountPerDeck)
        throws GameException {
```

- Replace the code that generates a SOAPFaultException with code that generates a GameException. This code:

```
if (numberOfDecks < 1 || numberOfDecks > 2) {
    try {
        QName faultCode = new
QName(SOAPConstants.URI_NS_SOAP_1_1_ENVELOPE, "Client");
        String reasonText = "Only 1-2 decks supported";
        SOAPFault fault =
SOAPFactory.newInstance().createFault(reasonText, faultCode);
        throw new SOAPFaultException(fault);
```

```
    } catch (SOAPException ex) {
        throw new IllegalArgumentException("Only 1-2 decks
supported");
    }
}
```

- Should change to

```
if (numberOfDecks < 1 || numberOfDecks > 2) {
    throw new GameException("Only 1-2 decks supported");
}
```

- Submit an out-of-range request using the RESTClient FireFox extension.
  - Launch the RESTClient FireFox extension.
  - Add a header, Content-Type: text/xml
  - Set the URL to http://localhost:7001/GenericCardGameWS/CardGameService.
  - Change the HTTP Method to POST.
  - Complete the body to indicate that you would like to build a game with three decks.

```
<env:Envelope
xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
  <env:Header />
    <env:Body>
    <createCardGame xmlns="http://games/">
      <deck-count xmlns="">3</deck-count>
      <jokers-per-deck xmlns="">2</jokers-per-deck>
    </createCardGame>
  </env:Body>
</env:Envelope>
```

  - You should receive a response with a 500 OK status code and a response body containing the GameException.
  - View the web services WSDL file at http://localhost:7001/GenericCardGameWS/CardGameService?wsdl.
  - Search for the word "fault" in the WSDL file. You should see that the createCardGame operation now has an input, an output, and a fault.
  - The XML Schema imported by the WSDL declares the structure of the GameException type: http://localhost:7001/GenericCardGameWS/CardGameService?xsd=2
  - By placing the fault information in the WSDL file, clients can be made aware that operations may fail for application-specific reasons.

10. Handle custom application SOAP faults in a JAX-WS client.
  - In NetBeans, open the IndianRummyWS project that functions as a JAX-WS client.
  - Remember that NetBeans copies a services WSDL file locally into the project when generating client references.
  - Expand the Web Service References branch in the IndianRummyWS project.
  - Right-click CardGameService and choose refresh.
  - Select the check box that says "Also replace the local wsdl file with the original wsdl located at" and click Yes.

- Open the RummyGameCollectionResource.java file. There should be a compiler error in the createCardGame method because port.createCardGame(deckCount, jokersPerDeck) throws a GameException_Exception.

```
private static String createCardGame(int deckCount, int
jokersPerDeck) {
    cards.CardGameService_Service service = new
cards.CardGameService_Service();
    cards.CardGameService port =
service.getCardGameServicePort();
    return port.createCardGame(deckCount, jokersPerDeck);
}
```

- Add a throws clause to the createCardGame method for the GameException_Exception exception.
- In the createGame method, modify the lines of code that call createCardGame to handle the GameException_Exception.

```
String gameId;
try {
    int players = rummyGame.getPlayers().size();
    if (players >= 2 && players <= 3) {
        gameId = createCardGame(1, 2);
    } else if (players >= 4 && players <= 6) {
        gameId = createCardGame(2, 2);
    } else if (players >= 7 && players <= 10) {
        gameId = createCardGame(3, 2);
    } else {
        rb = Response.status(Response.Status.BAD_REQUEST);
        return rb.build();
    }
} catch (GameException_Exception ex) {
    GameException ge = ex.getFaultInfo();
    rb = Response
            .status(Response.Status.BAD_REQUEST)
            .entity(ge.getMessage());
    return rb.build();
}
```

- Add any required imports.
- Save your changes and deploy the IndianRummyWS project.
- Launch the RESTClient FireFox extension.
- Using the http://localhost:7001/IndianRummyWS/resources/games URL, create a new game using the POST method with a body indicating that duke, larry, peter, james, mike, cindy, joe, and diganta wish to play a game.
  - There should be a header of Content-Type: application/xml.

    WARNING! The request body must not have any whitespace before the XML preamble.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<rummy-game>
  <player>
    <userName>duke</userName>
  </player>
  <player>
    <userName>larry</userName>
  </player>
  <player>
    <userName>peter</userName>
  </player>
  <player>
    <userName>james</userName>
  </player>
  <player>
    <userName>mike</userName>
  </player>
  <player>
    <userName>cindy</userName>
  </player>
  <player>
    <userName>joe</userName>
  </player>
  <player>
    <userName>diganta</userName>
  </player>
</rummy-game>
```

  - The JAX-WS client (RummyGameCollectionResource.java) is catching GameException_Exception and creating a 400 Bad Request HTTP response using the information contained in the exception.
- The primary difference between throwing a SOAPFaultException and a custom application is the same as the difference between checked and unchecked exceptions in Java. If a SOAP fault is added to an application, clients (at least JAX-WS clients) are forced to acknowledge the possibility of the fault and add in fault-handling code.

Practice 11-2: JAX-RS Error Handling

Overview

In this practice, you modify the IndianRummyWS RESTful web service to throw exceptions. RESTful web services that rely on HTTP use status codes to indicate errors and exceptions.

HTTP status codes in the 2xx range indicate a successful request. Codes in the 4xx range indicate some type of client error. Codes in the 5xx range indicate some type of error on the server side.

Some client requests result in HTTP responses with 4xx without any developer involvement. For example, a client request to a URL that does not correspond to a resource results in a 404 Not Found response.

To "throw" application exceptions from a JAX-RS application you return a response with a 4xx or 5xx status code. In this practice you will look at the different ways to return 4xx and 5xx responses.

Assumptions

You have completed Practice 11-1: JAX-WS Basic Error Handling.

Tasks

1. In NetBeans, open the IndianRummyRS project.
2. Open the RummyGameResource.java file and locate the getGame method.
3. At the beginning of the getGame method you should see code that checks if a there is a group for this game. If there is no group, it returns a 404 response.

```
ResponseBuilder rb;

if (!isGroupPresent("game-" + gameId)) {
    rb = Response.status(Response.Status.NOT_FOUND);
    return rb.build();
}
```

   - The use of a ResponseBuilder to return a Response with a 4xx or 5xx error code is effectively like throwing an exception. Response.Status.NOT_FOUND corresponds to a 404 numeric status.
4. Modify the getGame method to throw WebApplicationException instead of returning a Response object.
   - A WebApplicationException provides a means to use the Java throw statement and simplify the return type of a method.
   - The purpose of the getGame method is to get a rummy.jaxb.RummyGame instance and return it to the client. Modify the return type of the getGame method to be RummyGame.

```
@GET
public RummyGame getGame(@Context UriInfo uriInfo,
@PathParam("id") String gameId) {
```

   - Delete the line that declares the ResponseBuilder local variable.
   - Modify all lines that use a ResponseBuilder to return a 4xx or 5xx status code to throw a WebApplicationException. For example, this code:

```
if (!isGroupPresent("game-" + gameId)) {
```

```
    rb = Response.status(Response.Status.NOT_FOUND);
    return rb.build();
}
```

Would be modified as shown:

```
if (!isGroupPresent("game-" + gameId)) {
    throw new
WebApplicationException(Response.Status.NOT_FOUND);
}
```

- The final return statement in the getGame method can now simply return the game instance.
- Because WebApplicationException is a subclass of RuntimeException, no throws clause is required.
- Fix any imports and save your changes.
- The functionality of the getGame method remains exactly the same as before but the implementation is simplified (no more ResponseBuilder).

5. Throw a checked exception from a resource method.
   - In RummyGameResource.java, the getGame method uses a JAXBContext that potentially throws a JAXBException.

```
RummyGame game;
try {
    JAXBContext jaxbContext =
JAXBContext.newInstance(RummyGame.class);
    Unmarshaller u = jaxbContext.createUnmarshaller();
    String groupDesc = getGroupDescription("game-" + gameId);
    ByteArrayInputStream in = new
ByteArrayInputStream(groupDesc.getBytes());
    game = (RummyGame) u.unmarshal(in);
} catch (JAXBException ex) {
    logger.log(Level.SEVERE, "Failed to read game XML", ex);
    throw new
WebApplicationException(Response.Status.INTERNAL_SERVER_ERROR);
}
```

   - Modify the getGame method to throw a JAXBException.

```
@GET
public RummyGame getGame(@Context UriInfo uriInfo,
@PathParam("id") String gameId) throws JAXBException {
```

   - Remove the try-catch block and simplify the code using the JAXBContext.

```
JAXBContext jaxbContext =
JAXBContext.newInstance(RummyGame.class);
Unmarshaller u = jaxbContext.createUnmarshaller();
String groupDesc = getGroupDescription("game-" + gameId);
ByteArrayInputStream in = new
ByteArrayInputStream(groupDesc.getBytes());
```

```
RummyGame game = (RummyGame) u.unmarshal(in);
```

- Save your changes.
- You have simplified the code but what happens when a JAXBException is thrown?

6. Corrupt the XML for a game and cause a JAXBException to be thrown.
   - Start FireFox and launch the RESTClient extension.
   - Using the http://localhost:7001/IndianRummyWS/resources/games URL, create a new game using the POST method with a body indicating that duke and larry wish to play a game.
     - There should be a header of Content-Type: application/xml

       WARNING! The request body must not have any whitespace before the XML preamble.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<rummy-game>
  <player>
    <userName>duke</userName>
  </player>
  <player>
    <userName>larry</userName>
  </player>
</rummy-game>
```

   - The response status should be 201 Created. The Location header should be present and set to the URL of the created game. Copy the UUID at the end of the game resource URL to the clipboard.
   - http://localhost:7001/IndianRummyWS/resources/games/**ec62a19d-c3d5-4496-8a82-2fb598ef783e**

     Note: Your UUID is different.
   - Use the user-management-service web service to corrupt the description (metadata) of the game group. Open a web browser and visit http://localhost:7001/wls_utc/begin.do?wsdlUrl=http://localhost:7001/UserManager/user-management-service?wsdl.
   - Add the prefix "game-" to the UUID of the game and read the current description using the get-group-description operation. There should be an XML document inside the <return> tags of the response.
   - Continue using the WebLogic Test Client and call the set-group-description operation. Change the value of the game group to "Web Services are cool!" or any other non-XML text.
   - The RummyGameResource.getGame method should now throw a JAXBException when attempting to read the group description as XML.

7. Throw a checked exception from a JAX-RS resource method.
   - Using the RESTClient extension, attempt to get the representation of the game that was just corrupted.
   - Perform a GET request to http://localhost:7001/IndianRummyWS/resources/games/**ec62a19d-c3d5-4496-8a82-2fb598ef783e**

     Note: Your UUID is different and should match the one reported during task 6.

- You should receive a 500 Internal Server Error response with a response body that states "Internal Server Error."
- You should also see a long stack trace if you view the Oracle WebLogic Server log within NetBeans.
- Most exceptions thrown cause a 500 response but the response can be customized to any status and body.
- Leave the RESTClient window open as you will repeat the same request in a moment.

8. Add a custom exception mapping response handler to the IndianRummyWS project.
   - In NetBeans, open the IndianRummyWS project and create a Java class named RummyExceptionMapper in the rummy package.
   - The RummyExceptionMapper should be annotated with @Provider and implement the ExceptionMapper<EXCEPTIONTYPE> interface (where EXCEPTIONTYPE is the type of exception this mapper should convert to a response).

```
@Provider
public class RummyExceptionMapper implements
ExceptionMapper<JAXBException>
```

   - Implement the toResponse method that returns a Response for the method which threw the JAXBException.

```
@Override
public Response toResponse(JAXBException exception) {
    ResponseBuilder rb =
Response.status(Response.Status.INTERNAL_SERVER_ERROR);
    rb.type(MediaType.TEXT_PLAIN_TYPE);
    if(exception.getCause() != null) {
        rb.entity(exception.getCause().getMessage());
    } else if (exception.getMessage() != null) {
        rb.entity(exception.getMessage());
    } else {
        rb.entity("JAXB Problem, if this continues contact
support");
    }
    return rb.build();
}
```

   - Add any required import statements.
   - Open the RummyApplication.java file and modify the getSingletons method to produce a set that includes your exception mapper.
     - The initial size of the HashSet should now be 2.

```
HashSet<Object> set = new HashSet<>(2);
```

     - Create and add the exception mapper to the set.

```
set.add(new RummyExceptionMapper());
```

   - Add any required import statements, save all your changes, and redeploy the IndianRummyWS project.

9. Throw the mapped exception from a JAX-RS resource method.

- Switch back to the RESTClient extension and attempt to get the representation of the game that was just corrupted.
- Perform a GET request to http://localhost:7001/IndianRummyWS/resources/games/**ec62a19d-c3d5-4496-8a82-2fb598ef783e**

  Note: Your UUID is different and should match the one reported during task 6.
- You should receive a 500 Internal Server Error response but the raw response body now says "Content is not allowed in prolog" instead of "Internal Server Error."
- You should also no longer see a long stack trace if you view the Oracle WebLogic Server log within NetBeans.

10. Using an exception mapper you can convert any thrown exception to any response.
    - You may create your own subclass of Exception and throw it to custom exception mappers.
    - Custom exception mappers can generate any response, not just Internal Server Error. You might have a difficult time explaining to your peers why throwing an exception would ever result in a 200 OK response. Typically custom exception mappers respond with status codes in the 4xx and 5xx range.
    - Custom exception mappers also work well as a centralized place to log all exceptions thrown from resource methods.

Practices for Lesson 12: Java EE Security

Chapter 12

Practices for Lesson 12: Java EE Security

Practices Overview

In these practices, you will configure Java EE application security.

Practice 12-1: Enabling Authentication

Overview

In this practice you restrict HTTP-based access to URL patterns. Because the primary means of communication for web services is HTTP, securing HTTP transport channels is often the first step in securing web services. By itself, authentication does not secure a web service, so you at least require encrypted (TLS) connections.

Assumptions

You have completed Practice 11-2: JAX-RS Error Handling.

Tasks

1. In NetBeans, open the IndianRummyWS and RummyRules projects.
2. Delete the index.jsp file from the IndianRummyWS project.
3. Copy the index.jsp and DejaVuSans.tff files from the RummyRules project to the IndianRummyWS project.
4. Close the RummyRules project.
5. Currently any unauthenticated client can call operations on every resource (URL) in the IndianRummyWS project.
6. Open the web.xml configuration file in the IndianRummyWS project and view the source (instead of the form-based designer).

7. Create a minimal security configuration in web.xml that blocks all access to unauthenticated users.
   - Add the following elements after the existing <session-config>.
   - Restrict all paths and HTTP methods to player and admin roles.

```
<security-constraint>
    <display-name>Indian Rummy Root Path</display-name>
    <web-resource-collection>
        <web-resource-name>everything</web-resource-name>
        <url-pattern>/*</url-pattern>
    </web-resource-collection>
   <auth-constraint>
        <description/>
        <role-name>player</role-name>
        <role-name>admin</role-name>
   </auth-constraint>
</security-constraint>
```

   - Configure the application to use HTTP basic authentication.

```
<login-config>
    <auth-method>BASIC</auth-method>
</login-config>
```

- Declare the player and admin roles as roles that may be used throughout the web application (and should be mapped to principals).

```
<security-role>
    <description>A person that is allowed to play
rummy.</description>
    <role-name>player</role-name>
</security-role>
<security-role>
    <description>A person that is allowed to perform
administration tasks.</description>
    <role-name>admin</role-name>
</security-role>
```

8. Map Java EE roles to WebLogic principals in the weblogic.xml file.
   - Application servers support the idea of vendor-specific deployment descriptors that can be used to map roles to users and groups.
   - Map the player role to the players group.
   - Map the admin role to the weblogic user.

```
<?xml version="1.0" encoding="UTF-8"?>
<weblogic-web-app
xmlns="http://xmlns.oracle.com/weblogic/weblogic-web-app"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd
http://xmlns.oracle.com/weblogic/weblogic-web-app
http://xmlns.oracle.com/weblogic/weblogic-web-app/1.4/weblogic-
web-app.xsd">
  <jsp-descriptor>
    <keepgenerated>true</keepgenerated>
    <debug>true</debug>
  </jsp-descriptor>
  <context-root>/IndianRummyWS</context-root>
  <security-role-assignment>
      <role-name>player</role-name>
      <principal-name>players</principal-name>
  </security-role-assignment>
  <security-role-assignment>
      <role-name>admin</role-name>
      <principal-name>weblogic</principal-name>
  </security-role-assignment>
  <library-ref>
    <library-name>Jersey-1.17</library-name>
    <specification-version>1.17</specification-version>
    <implementation-version>1.17.0</implementation-version>
    <exact-match>true</exact-match>
```

```
  </library-ref>
</weblogic-web-app>
```

9. Save all changes. Every time you save changes to these files the application should automatically be redeployed if it was already deployed. Sometimes it takes a moment for WebLogic Server to finish redeploying the application. While testing if you are getting an unexpected result, wait for a few seconds (not minutes) and try again.
10. Open a command prompt and use the `cd` command to change to the D:\labs\student\resources directory.
    - Because web browsers retransmit authentication credentials under certain conditions it is less error prone to use the cURL command-line utility to test your authentication configuration.
    - You can use the RESTClient extension to add the Authorization header with the values correctly encoded for HTTP Basic authentication, but even after removing the header the Authentication header continues to be transmitted by FireFox.
11. Request the index.jsp page from the IndianRummyWS application. We are primarily interested in what the HTTP response status is, so the –head option can be used to only display headers.

```
curl -v -X GET http://localhost:7001/IndianRummyWS/index.jsp
```

    - The response for an authenticated user should be:

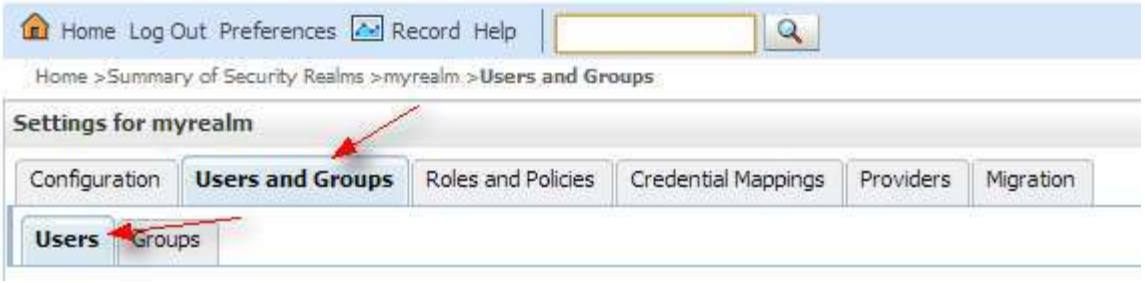
```
HTTP/1.1 401 Unauthorized
Date: Mon, 11 Mar 2013 18:28:31 GMT
Content-Length: 1468
Content-Type: text/html; charset=UTF-8
WWW-Authenticate: Basic realm="weblogic"
X-Powered-By: Servlet/3.0 JSP/2.2
```

    - Those in the admin role (the weblogic user) should be able to view the page. Use the -u option to specify a username of weblogic. When prompted, enter the welcome1 password.

```
curl -v -X GET -u weblogic
http://localhost:7001/IndianRummyWS/index.jsp
You should receive a 200 OK response.
HTTP/1.1 200 OK
Date: Mon, 11 Mar 2013 18:32:23 GMT
Content-Length: 9092
Content-Type: text/html; charset=UTF-8
Set-Cookie:
JSESSIONID=Gn92R2jXhQnKL5Ly9q422BmyTpNjpfjsDTc8G2djSk2HcFLppMp5!
825790820; path=/; HttpOnly
X-Powered-By: Servlet/3.0 JSP/2.2
```

12. View the current users and groups using the WebLogic administration console.
    - In a web browser, go to http://localhost:7001/console.
    - Log in with the weblogic username and welcome1 password.
    - Click Security Realms and then select the myrealm realm.

- Display the Users tab for the myrealm realm.



- You should see a list of users. The list will vary depending on which players you have created. Note that only ten users are shown at a time; use the Next and Previous links to navigate the pages of users.
- View the Groups tab. If you have any valid games still in existence you should see some groups named "game-UUID" where UUID is a long unique value.
- You can map roles to the users and groups in this realm.
- The user-management-service web service contains a custom EJB that is able to provision users and groups within WebLogic Server. There is no standard Java EE API to create a user or group that can then be used by Java EE declarative security.
- Find a user that belongs to the players group by clicking a user's name and view the Groups tab for that user. Possible users include duke, matt, tom, peter, cindy, diganta, james, joe, mike, and larry.

13. Using cURL, display the list of players. Authentication should be require to perform the action.
    - Because the JSON output should be formatted (Practice 10-5) it is easier to read the JSON output.

```
curl -v -X GET -H "Accept: application/json" -u weblogic
http://localhost:7001/IndianRummyWS/resources/players
```

14. Use an account that can function in the player role to make an authenticated request for index.jsp.
    - Because of the settings in weblogic.xml all members of the player**s** group can function in the player role.

```
curl -v -X GET -u joe
http://localhost:7001/IndianRummyWS/index.jsp
```

    - You should receive a 200 OK response if you supply a value player and password.

```
HTTP/1.1 200 OK
Date: Mon, 11 Mar 2013 20:20:03 GMT
Content-Length: 9092
Content-Type: text/html; charset=UTF-8
Set-Cookie:
JSESSIONID=PbpTR28T14BCvvv6NpT8zM0ZbMm3rzkkz9hhJCpX4fHQN2GhYw3R!
1569718065; path=/; HttpOnly
X-Powered-By: Servlet/3.0 JSP/2.2
```

15. Switch back to editing the web.xml file in NetBeans.
16. Add a security constraint that allows unauthenticated users to GET the WADL file for the application.
    - The automatically generated WADL file for Jersey JAX-RS applications is /resources/application.wadl.
    - It is a very important step for JAX-WS applications. The WSDL file is often required by tools that do not support authentication. If developers must be authenticated to access a WSDL file they may have to use a browser to download a local copy and generate client stubs using a local copy of the WSDL.

```
<security-constraint>
    <display-name>Indian Rummy WADL</display-name>
    <web-resource-collection>
        <web-resource-name>games</web-resource-name>
        <description>Game resource hierarchy</description>
        <url-pattern>/resources/application.wadl</url-pattern>
        <http-method>GET</http-method>
    </web-resource-collection>
</security-constraint>
```

    - Save your changes and allow WebLogic Server to redeploy the application.
17. Using cURL, perform a GET of the WADL file.

```
curl -v -X GET
http://localhost:7001/IndianRummyWS/resources/application.wadl
```

    - You should receive a 200 OK response.

```
HTTP/1.1 200 OK
Date: Mon, 11 Mar 2013 20:30:27 GMT
Content-Length: 5044
Content-Type: application/vnd.sun.wadl+xml
Last-Modified: Mon, 11 Mar 2013 15:30:26 CDT
```

```
X-Powered-By: Servlet/3.0 JSP/2.2
Vary: Accept
```

18. Switch back to editing the web.xml file in NetBeans.
19. Restrict all DELETE operations to those in the admin role.
    - The following resources respond to DELETE requests:
      - /resources/players/{id}
      - /resources/games
      - /resources/games/{id}
    - Because a security constraint that allows both player and admin roles to call any HTTP method for any path already exists (task 7) this new constraint must be more specific by either having a more specific path or using no wildcards.

```
<security-constraint>
    <display-name>Indian Rummy DELETE</display-name>
    <web-resource-collection>
        <web-resource-name>games</web-resource-name>
        <description>Game resource hierarchy</description>
        <url-pattern>/resources/*</url-pattern>
        <http-method>DELETE</http-method>
    </web-resource-collection>
    <auth-constraint>
        <description/>
        <role-name>admin</role-name>
    </auth-constraint>
</security-constraint>
```

    - Because /resources/* is a longer path than /*, this security constraint is the only one that applies to DELETE operations under the /resources/* path.
20. Using cURL, test the DELETE restriction.
    - Try to delete a user while not authenticated.

```
curl -v -X DELETE
http://localhost:7001/IndianRummyWS/resources/players/joe
```

    - You should receive a 401 Unauthorized response similar to:

```
HTTP/1.1 401 Unauthorized
Connection: close
Date: Mon, 11 Mar 2013 20:43:25 GMT
Content-Length: 1468
Content-Type: text/html; charset=UTF-8
WWW-Authenticate: Basic realm="weblogic"
X-Powered-By: Servlet/3.0 JSP/2.2
```

    - Try to delete a player while authenticated as another player.

```
curl -v -X DELETE -u matt
http://localhost:7001/IndianRummyWS/resources/players/joe
```

    - You should receive a 403 Forbidden response similar to:

```
HTTP/1.1 403 Forbidden
Connection: close
Date: Mon, 11 Mar 2013 20:43:43 GMT
Content-Length: 1166
Content-Type: text/html; charset=UTF-8
X-Powered-By: Servlet/3.0 JSP/2.2
```

- Try to delete a player while authenticated as the weblogic user who is in the admin role.

```
curl -v -X DELETE -u weblogic
http://localhost:7001/IndianRummyWS/resources/players/joe
```

- You should receive a 200 OK response similar to:

```
HTTP/1.1 200 OK
Connection: close
Date: Mon, 11 Mar 2013 20:44:10 GMT
Content-Length: 0
Content-Type: application/xml
Set-Cookie:
JSESSIONID=QK2JR2ChgJGl8JhZ10sLCZKh83DvfLK59S0y3lPDm22JzlhNl54y!
1569718065; path=/; HttpOnly
X-Powered-By: Servlet/3.0 JSP/2.2
```

- Make sure the admin restriction is applied only to DELETE requests by performing a GET request to the players resource as a non-admin user.

```
curl -v -X GET -u matt
http://localhost:7001/IndianRummyWS/resources/players
```

- You should receive a 200 OK response similar to:

```
HTTP/1.1 200 OK
Date: Mon, 11 Mar 2013 20:51:56 GMT
Content-Length: 1346
Content-Type: application/xml
Set-Cookie:
JSESSIONID=p2B4R2DMDybXhfzKxGyKpGkSqs1zpPckJWRr1vGL6gr9XmsHqrtL!
1569718065; path=/; HttpOnly
X-Powered-By: Servlet/3.0 JSP/2.2
```

21. Switch back to editing the web.xml file in NetBeans.
22. Allow unauthenticated users to join.
    - If you have to be authenticated to create an account, non-members cannot join the service themselves. That might work for an invite-only system but our Indian Rummy web service should be open to the public.
    - Unauthenticated users should be able to make a PUT request to /resources/players/{ID}.

```
<security-constraint>
    <display-name>Indian Rummy Add Player</display-name>
    <web-resource-collection>
        <web-resource-name>Add player PUT and OPTIONS</web-
resource-name>
```

```
        <description>Navigate to resource</description>
        <url-pattern>/resources/players/*</url-pattern>
        <http-method>PUT</http-method>
    </web-resource-collection>
</security-constraint>
```

23. Using cURL, test the rules to create a player.
    - Create a new player named caesar.
      - Use a URL of http://localhost:7001/IndianRummyWS/resources/players/caesar.
      - Escape the quotes in the JSON content with backslashes.

```
curl -v -H "Content-Type: application/json" -X PUT --data
"{\"userName\":\"caesar\",\"password\":\"welcome1\"}"
http://localhost:7001/IndianRummyWS/resources/players/caesar
```

    - You should receive a 201 Created response similar to:

```
HTTP/1.1 201 Created
Date: Mon, 11 Mar 2013 23:35:06 GMT
Location:
http://localhost:7001/IndianRummyWS/resources/players/caesar
Content-Length: 0
Content-Type: application/xml
X-Powered-By: Servlet/3.0 JSP/2.2
```

Practice 12-2: Enabling Confidentiality

Overview

In this practice, you enable HTTPS (SSL/TLS) connections.

Enabling authentication by itself is insecure. Many utilities, such as WireShark, can capture and display network packets. Requiring authentication credentials does not increase security if it is trivial for someone else to read those credentials as they are in transit.

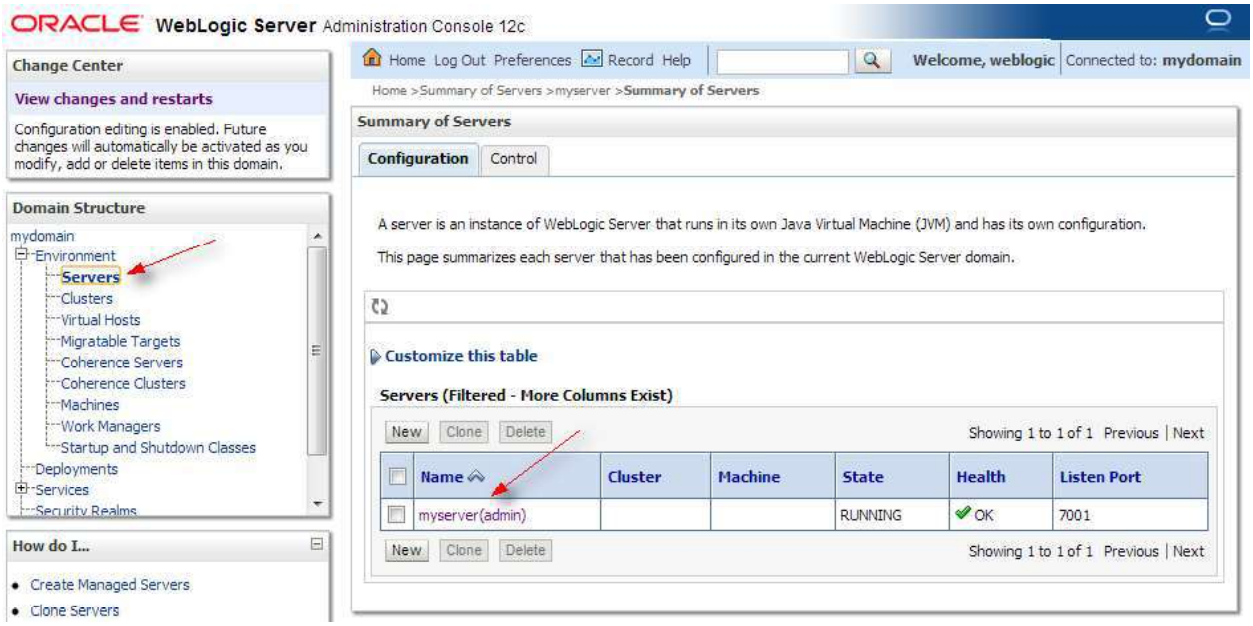
Use HTTPS ensures that if someone does capture your network traffic they are not be able to understand the content.

Assumptions

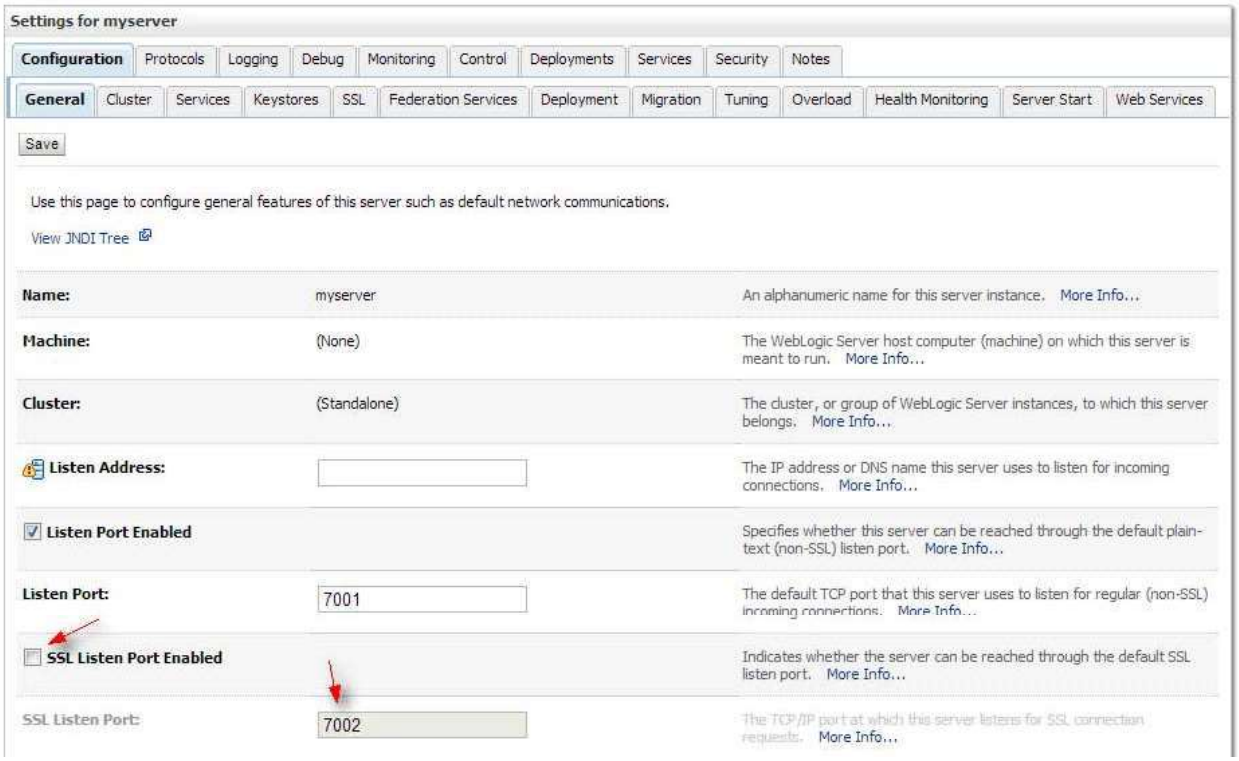
You have completed Practice 12-1: Enabling Authentication.

Tasks

1. Enable HTTPS using the WebLogic administration console.
   - In a web browser, go to http://localhost:7001/console.
   - Log in with the weblogic username and welcome1 password.
   - Click Environment > Servers and then select the myserver(admin) server.

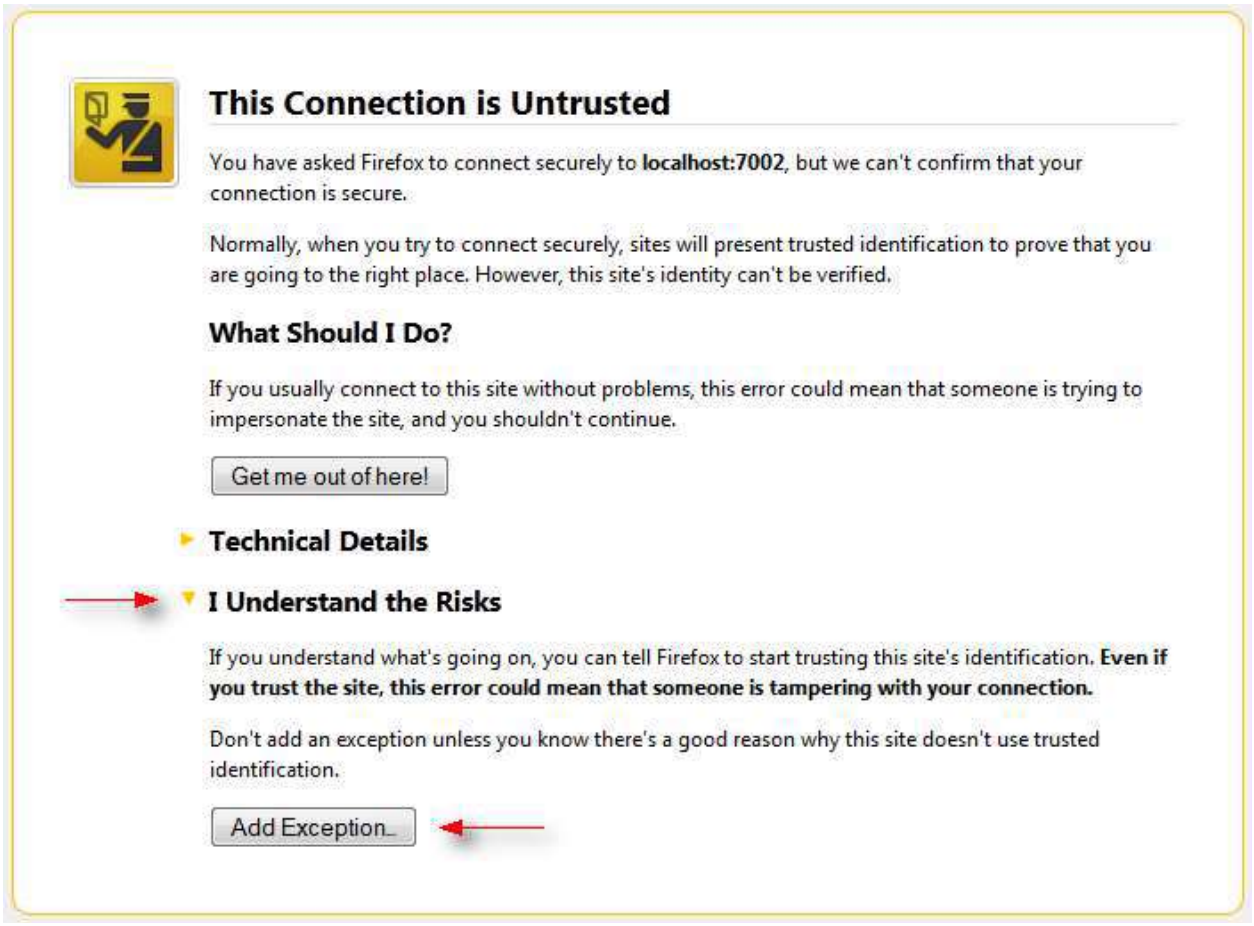


- Under Configuration > General, select the check box named SSL Listen Port Enabled.



- Click the Save button at the bottom of the page.
- This causes WebLogic Server to listen for HTTPS connection on port 7002.

  Note: WebLogic Service includes a demonstration certificate for testing purposes. If this was a production server you would purchase a signed certificate. Because the demonstration certificate is not signed by a trusted certificate authority (CA) it cannot be used by clients to verify the identity of the server. It can still be used to create an encrypted connection between a client and the server. When connecting to port 7002 using HTTPS you have to add an exception to the FireFox web browser and Internet Explorer may refuse to connect to the server on port 7002.
- Open FireFox web browser and go to https://localhost:7002/IndianRummyWS/.
- You should be greeted with a warning message. Expand the I Understand the Risks node and click the Add Exception button.



Note: As the screen says, you should add exceptions like these only if you understand why you are getting them and if they are expected.

The RESTClient extension is not able to make HTTPS connections to your server unless you add the exception. Remember that some URLs have a security constraint that redirect to HTTPs. These redirects cause RESTClient to fail to load any responses unless the certificate exception has been added.

- In the Add Security Exception dialog box, click Confirm Security Exception.
- After supplying a valid username and password you should see a web page describing the rules of Indian Rummy.
- Test the HTTPS connection using cURL. The –k option is used to accept the self-signed certificate.

```
curl -k -v -X GET
https://localhost:7002/IndianRummyWS/resources/application.wadl
```

2. Using the web.xml deployment descriptor, configure all resource URLs except the WADL URL to require HTTPS.
   - Add a <user-data-constraint> element with a child element of <transport-guarantee> that has a value of CONFIDENTIAL.

```
<user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
</user-data-constraint>
```

- Every <security-constraint> except the WADL constraint should be configured to require HTTPS.
- Save your changes and allow WebLogic Server to redeploy your application.
- Test the changes using cURL.

```
curl -k -v -X GET –u matt
http://localhost:7001/IndianRummyWS/resources
```

- You should receive a response that indicates that the content is now located elsewhere. Note the Location header that points to the new location.

```
HTTP/1.1 302 Moved Temporarily
Connection: close
Date: Tue, 12 Mar 2013 00:34:38 GMT
Transfer-Encoding: chunked
Location: https://localhost:7002/IndianRummyWS/resources
X-Powered-By: Servlet/3.0 JSP/2.2
```

- Many applications such as FireFox and the RESTClient extension automatically issue a second request to the new location when receiving a 302 response.
- To follow redirects when using cURL (and continue to supply your username and password) to every location you are redirected to add the –location-trusted option.

```
curl -k --location-trusted -v -X GET -u matt
http://localhost:7001/IndianRummyWS/resources
```

- Back at your work environment you may follow a different process to enable HTTPS. The configuration steps shown here are for when clients communicate directly with an application server.

Practices for Lesson 13: Securing JAX-WS Services

Chapter 13

Practices for Lesson 13: Securing JAX-WS Services

Practices Overview

In these practices, you will generate security keys and certificates, reconfigure WebLogic Server to support WS-Security, and create and consume a WS-Security enabled web service.

Practice 13-1: Securing a JAX-WS Endpoint with WS-Security

Overview

In this practice you use WS-Security to enhance the security of a JAX-WS web service. You will add extra security to the GenericCardGameWS project. The CardGameService web service provides the foundation to creating card games, including multiplayer games. The messages it receives and responds with would be a high-value target for players attempting to view the hands of other players.

Remember, WS-* extensions are not a mandatory element of JAX-WS and they are primarily geared toward non-functional requirements. You will find that the method(s) used by application servers to enable and use these features (if they are supported at all) vary. In this practice we use the most developer-centric method (code modification) to request that policies be applied. WebLogic Server also supports WS-Policy configuration via the administration console.

Tasks

1. Generate SSL keys and certificates.
   - WS-Security requires stronger SSL keys than are provided by the demo certificates.
   - In a command window run D:\labs\student\exercises\lab13\certgen.bat.
   - The full list of keytool commands used to create SSL certificates can be found in Practice A-2: Generating Security Certificates and Keys.
2. Configure SSL and Web Service Security Keys.
   - This step configures WS-Security within WebLogic Server.
   - In a command window run D:\labs\student\exercises\lab13\sslconfig.bat
   - The full list of steps required to configure WS-Security is listed in Practice A-3: Specifying SSL Keys in WebLogic Server.
   - Start WebLogic Server and use the FireFox web browser to connect to the Admin Console using HTTP. Approve any security exceptions.

     ```
     https://localhost:7002/console
     ```

   - The sslconfig.bat program shut down the WebLogic Server instance. You may need to refresh the Oracle WebLogic Server node in the NetBeans Service tab to see the current state.
     - Open NetBeans.
     - Switch to the Services tab.
     - Expand the Servers node.
     - If Oracle WebLogic Server still shows as being in a Running state (green triangle) then refresh the status by right-clicking Oracle WebLogic Server and choosing Refresh.
   - Start Oracle WebLogic Server using the NetBeans Service tab.
3. In NetBeans, open the GenericCardGameWS project you last worked on during Practice 11-1: JAX-WS Basic Error Handling.
4. Enable JAX-WS HTTP dumping.
   - If you did not perform "Optional Practice 5-7: Using WS-MakeConnection with a JAX-WS Client" then perform the steps below to enable JAX-WS server message dumping.
   - Open the Tools > Servers menu in NetBeans.
   - Select Oracle WebLogic Server.

- Open the Platform tab.
- Add a VM Options value of:

```
-Dcom.sun.xml.ws.transport.http.HttpAdapter.dump=true
```

- Click Close.
- Use the Service tab in NetBeans to restart WebLogic Server. After restarting all requests and responses to JAX-WS, endpoints should appear in the WebLogic output window.

5. To enable the use of the WebLogic WS-Policy annotations, create a new library in NetBeans.
   - Open the Tools menu and choose Ant Libraries.
   - Search for the WebLogic Web Services API library. If you have already created it you may skip to task 6.
   - Create a new library.
     - Name: WebLogic Web Services API
     - Library Type: Server Libraries
   - While WebLogic Web Services API is selected, click the Add JAR/Folder button and add D:\weblogic\wls\modules\ws.api_2.0.0.0.jar.
   - Click OK.
6. Add the WebLogic Web Services API library to the GenericCardGameWS project.
7. Open the CardGameService.java file.
8. View the Wssp1.2-2007-Wss1.1-UsernameToken-Plain-X509-Basic256.xml policy file that you will be applying.
   - In the GenericCardGameWS project, expand Libraries > Oracle WebLogic Server > weblogic.jar > weblogic.wsee.policy.runtime
   - Double-click the Wssp1.2-2007-Wss1.1-UsernameToken-Plain-X509-Basic256.xml file.
   - The WS-SecurityPolicy elements you see in the file are only some of the elements that can be used when crafting a WS-Security policy. Many of the elements can be mixed to form different configurations. To ensure maximum compatibility it is recommended that you use a prewritten policy.
   - Close the policy file.
9. Add a @Policy annotation to the CardGameService class that applies the Wssp1Wssp1.2-2007-Wss1.1-UsernameToken-Plain-X509-Basic256.xml policy and attaches the policy to the generated WSDL file.

```
@Policy(uri = "policy:Wssp1.2-2007-Wss1.1-UsernameToken-Plain-
X509-Basic256.xml", attachToWsdl = true)
```

   - The @Policy annotation requires the following import statement.

```
import weblogic.jws.Policy;
```

10. Using a @Policies annotation with nested @Policy annotations requires that message bodies be signed and encrypted. Apply the annotations at the method level to all public methods.

```
@Policies({
    @Policy(uri = "policy:Wssp1.2-2007-SignBody.xml",
attachToWsdl = true),
    @Policy(uri = "policy:Wssp1.2-2007-EncryptBody.xml",
attachToWsdl = true)})
```

- Add any required imports.

11. Save your changes and deploy the GenericCardGameWS project.
12. View the WS-Policy additions to the CardGamesService WSDL file by using the RESTClient extension.
    - Launch the RESTClient FireFox extension.
    - Enter http://localhost:7001/GenericCardGameWS/CardGameService?wsdl in the URL form field and click SEND.
    - View the Response Body (Highlight). You should see a <wsp1_2:Policy> element near the beginning of the WSDL file that has been added.
    - Close the RESTClient extension.
    - You will not be able to use RESTClient or the WebLogic web-based Test Client to test the secured service. You will observe the messages as they are dumped to the output window.
13. Open the IndianRummyWS project in NetBeans.
14. Add a static cards.CardGameService getGameService() method to a new ServiceFactory class.
    - View the RummyGameResource.java and RummyGameCollectionResource.java files.
    - All the methods inserted by the Insert Code > Call Web Service Operation tool retrieve a service and port locally within their method body. Instead of adding WS-Security client code to every method you will refactor port creation into a new class.
    - Create the rummy.ServiceFactory Java class.
    - Add a logger instance.

```
private static final Logger logger = Logger.getLogger("rummy");
```

    - Create the getGameService method.

```
public static CardGameService getGameService() {

}
```

        - In the getGameService method, obtain the service and port for the card game service.

```
CardGameService_Service service = new CardGameService_Service();
CardGameService port = service.getCardGameServicePort();
Create the string variables for keystore and certificate paths,
aliases, and passwords.
String serverCertFile = "D:/labs/student/keystores/server.pem";
String clientKeyStore = "D:/labs/student/keystores/client.jks";
String clientKeyStorePass = "welcome1";
String clientKeyAlias = "client";
String clientKeyPass = "welcome1";
```

        - Create a list to store the username and certificate credential providers in.

```
List credProviders = new ArrayList();
```

        - Create and store the username credential provider.

```
ClientUNTCredentialProvider unt =
```

```
        new ClientUNTCredentialProvider("weblogic".getBytes(),
"welcome1".getBytes());
credProviders.add(unt);
```

- Create and store the certificate credential provider.

```
try {
    final X509Certificate serverCert =
            (X509Certificate)
CertUtils.getCertificate(serverCertFile);
    serverCert.checkValidity();

    CredentialProvider cp =
            new ClientBSTCredentialProvider(clientKeyStore,
clientKeyStorePass,
            clientKeyAlias, clientKeyPass,
            "JKS", serverCert);
    credProviders.add(cp);
} catch (Exception ex) {
    logger.log(Level.SEVERE, "Failed to load binary security
token", ex);
}
```

- Get the request context and add the credential providers to it.

```
Map<String, Object> requestContext =
        ((BindingProvider) port).getRequestContext();

requestContext.put(WSSecurityContext.CREDENTIAL_PROVIDER_LIST,
        credProviders);
```

- Add a trust manager that trusts all certificates to the request context.

```
requestContext.put(WSSecurityContext.TRUST_MANAGER,
        new TrustManager() {
            @Override
            public boolean certificateCallback(X509Certificate[]
chain, int validateErr) {
                return true;
            }
        });
```

- Return the port.

```
return port;
```

- Add any required import statements.

15. Modify the RummyGameResource and RummyGameCollectionResource classes to use the new port factory.
    - Open the RummyGameResource.java file.
    - Replace every occurrence of cards.CardGameService_Service:

```
service = new cards.CardGameService_Service();
cards.CardGameService port = service.getCardGameServicePort();
```

- With a call to the port factory:

```
cards.CardGameService port = ServiceFactory.getGameService();
```

- Open the RummyGameCollectionResource.java file and repeat the process.
- Save all your changes.

16. Refresh the JAX-WS artifacts and cached WSDL.
    - Expand the Web Service References node in the IndianRummyWS project.
    - Right-click CardGameService and choose Refresh.
    - Select the check box labeled "Also replace local wsdl file with original located at".
    - Click the Yes button.
17. Deploy and run the IndianRummyWS project.
    - Save any remaining changes.
    - Deploy the project.
    - Clear the WebLogic Server output window.
      - Select the Output tab that should be shown in the bottom panel of NetBeans.
      - Select the Oracle WebLogic Server subtab.
      - Right-click in the Oracle WebLogic Server output window and choose Clear.

        Note: If the Oracle WebLogic Server output window is missing you can right-click the Oracle WebLogic Server instance under the Services tab and select View Server Log.
    - Use the RESTClient extension to test the Indian Rummy RESTful resources (which are clients of the WS-Security enabled service).
    - Add a header of Content-Type: application/xml
    - Check the listing of current players by making a GET request to https://localhost:7002/IndianRummyWS/resources/players. If needed, add at least two players by making a PUT request to https://localhost:7002/IndianRummyWS/resources/players/PLAYERNAME. (replace PLAYERNAME with the player name)

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<player>
  <userName>tom</userName>
  <password>welcome1</password>
</player>
```

    - Create a game with players that are valid for your server by making a POST request to https://localhost:7002/IndianRummyWS/resources/games.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<rummy-game>
  <player>
    <userName>tom</userName>
  </player>
  <player>
    <userName>matt</userName>
  </player>
```

```
</rummy-game>
```

- Inspect the secured SOAP messages.
  - View the Oracle WebLogic Server output window in NetBeans.
  - The secured messages are too long to display by default. Right-click in the output window and select Wrap text.
  - Locate one of the SOAP requests to the CardGameService by using the SOAPAction header.
  - Highlight the XML request body and copy it to the clipboard.
  - Switch to the Favorites tab.
  - Expand the Exercises folder.
  - Right-click the lab13 folder and choose New > Empty File.
  - Name the file soap-request.xml.
  - Paste the SOAP XML into the file.
  - Right-click the newly created file and choose Format.
  - Locate the <S:Body> element. Can you identify the game ID contained in this message anymore?
  - Repeat the process for the corresponding SOAP response.

(Optional) Practice 13-2: Improving the Performance of JAX-WS Clients

Overview

In this practice, you improve the performance of your JAX-WS clients in the IndianRummyWS project. JAX-WS client proxies and ports are expensive to create. By caching and reusing them you can greatly improve performance.

Assumptions

You have completed Practice 13-1: Securing a JAX-WS Endpoint with WS-Security that created a factory for the CardGameService port.

Tasks

1. Use the RESTClient extension to create several new games of Indian Rummy and track the time it takes to build games.
   - Open the RESTClient extension.
   - Add a header of Content-Type: application/xml
   - Check the listing of current players by making a GET request to https://localhost:7002/IndianRummyWS/resources/players. If needed add at least two players by making a PUT request to https://localhost:7002/IndianRummyWS/resources/players/PLAYERNAME. (replace PLAYERNAME with the player name)

     ```
     <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
     <player>
       <userName>tom</userName>
       <password>welcome1</password>
     </player>
     ```

   - Create a game with players that are valid for your server by making a POST request to https://localhost:7002/IndianRummyWS/resources/games.

     ```
     <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
     <rummy-game>
       <player>
         <userName>tom</userName>
       </player>
       <player>
         <userName>matt</userName>
       </player>
     </rummy-game>
     ```

   - Create several games and average the time it takes to make a game.
2. Modify the ServiceFactory class to prebuild the service using a local copy of the WSDL file.
   - Add a static field to hold the prebuilt service instance.

     Note: The JAX-WS specification does not guarantee the thread-safety of service or proxy instances. Depending on your JAX-WS implementation you might have to synchronize access to shared instances of service and proxy objects.

```
private static CardGameService_Service gameService;
```

- Use a static code block to initialize the service field. When building a service instance, use the local copy of the WSDL file that was added to your project by wsimport.

```
static {
    final String wsdlPath =
"../../wsdl/localhost_7001/GenericCardGameWS/CardGameService.wsd
l";
    try {
        URL url = ServiceFactory.class.getResource("");
        url = new URL(url, wsdlPath);
        gameService = new CardGameService_Service(url);
    } catch (Exception e) {
        logger.warning("Failed to create URL for the wsdl
Location: " + wsdlPath + ", retrying without local file");
        logger.warning(e.getMessage());
        gameService = new CardGameService_Service();
    }
}
```

- Comment out the creation of the service at the beginning of the getGameService method.

3. Modify the ServiceFactory to cache a CardGameService instance using a ThreadLocal variable.
   - Add a ThreadLocal<CardGameService> field to the ServiceFactory class.

     Note: A ThreadLocal provides the ability to store and retrieve an object in a way such that only the thread that stored the object can retrieve it.

```
private static final ThreadLocal<CardGameService> localGamePort
= new ThreadLocal<>();
```

   - Remove the existing line that creates a port in the getGameService method.

```
//CardGameService port = service.getCardGameServicePort();
```

   - The remaining lines are placed in the else block of an if-else statement.
   - In the beginning of the getGameService method add the if-else statement that uses ThreadLocal to get and set the port.

```
CardGameService port = localGamePort.get();
if (port != null) {
    return port;
} else {
    port = gameService.getCardGameServicePort();
    localGamePort.set(port)
    // existing code here
}
```

4. Modify the ServiceFactory to provide a caching factory method for UserManagementType ports.
   - Add the required fields.

```
private static UserManagementService userService;
private static final ThreadLocal<UserManagementType>
localUserPort = new ThreadLocal<>();
```

- Add a static block to initialize the service using a local copy of the WSDL file.

```
static {
    final String wsdlPath =
"../../wsdl/localhost_7001/UserManager/user-management-
service.wsdl";
    try {
        URL url = ServiceFactory.class.getResource("");
        url = new URL(url, wsdlPath);
        userService = new UserManagementService(url);
    } catch (Exception e) {
        logger.warning("Failed to create URL for the wsdl
Location: " + wsdlPath + ", retrying without local file");
        logger.warning(e.getMessage());
        userService = new UserManagementService();
    }
}
```

- Add a getUserService method that leverages the ThreadLocal variable to cache port instances.

```
public static UserManagementType getUserService() {
    UserManagementType port = localUserPort.get();
    if (port != null) {
        return port;
    } else {
        port = userService.getUserManagementPort();
        localUserPort.set(port);
        return port;
    }
}
```

- Add any required import statements and save your changes.

5. Modify the PlayersResource, RummyGameCollectionResource, and RummyGameResource classes to use the ServiceFactory.
   - Replace all occurrences of:

```
users.UserManagementService service = new
users.UserManagementService();
users.UserManagementType port = service.getUserManagementPort();
```

   with:

```
users.UserManagementType port = ServiceFactory.getUserService();
```

6. Save all your changes and deploy the project.
7. Repeat task 1 to observe the performance improvement.

Practices for Lesson 14: Securing JAX-RS Services with Jersey

Chapter 14

Practices for Lesson 14: Securing JAX-RS Services with Jersey

Practices Overview

In these practices, you will secure a RESTful web service.

Practice 14-1: Using Java EE Roles and Principles

Overview

In this practice you use the @RolesAllowed annotation to implement declarative security. You will also use programmatic security in places where declarative security cannot be used to determine authorization.

@RolesAllowed, @PermitAll, @DenyAll, and @RunAs are just some of the annotations from JSR-250 Common Annotations for the Java Platform. These common annotations can be used in multiple types of Java EE components such as JAX-WS EJB endpoints. In this practice you will apply them to JAX-RS endpoints.

These annotations are used to declare security constraints and configure declarative security for your Java EE application. In some cases, declarative security is not enough. For example, take a method like deleteUser(String userId). If only someone in the "admin" role can call this method then you can use declarative security. What if users are allowed to delete their own account? In this case you must allow all users to call the method (using declarative security to make sure the caller was a valid user) but write code in the deleteUser method to determine if the account being deleted belonged to the person calling the method.

Assumptions

Whereas this practice assumes you have completed Practice 13-1: Securing a JAX-WS Endpoint with WS-Security, the actions you perform in this practice are more dependent on Practice 12-1: Enabling Authentication and Practice 12-2: Enabling Confidentiality.

In order to be able to check if a caller is in the correct role they must have authenticated to the application. For JAX-RS applications you will typically use the web.xml deployment description to force an authentication challenge. HTTP Basic authentication is the simplest form and one of the most commonly used methods of authentication but it transmits credentials in a non-encrypted form. To securely use HTTP Basic authentication you should only transmit credentials when using HTTPS (SSL) connections.

Tasks

1. In NetBeans, open the IndianRummyWS project if it is not already open.
2. Open the web.xml configuration file and review the security constraints that are in place for the web application.
3. Open the weblogic.xml configuration file and review the role mapping. Remember, when using @RolesAllowed({"role1", "role2"}), the role names are not the user or group names, they are the roles you defined in your web.xml and mapped in your weblogic.xml deployment descriptors.
4. Configure Jersey to support security annotations.
   - Jersey does not support the common security annotation unless configured correctly.
   - Open the web.xml configuration file.
   - Below the <session-config> element add the servlet and servlet-mapping elements to configure the Jersey servlet with the RolesAllowedResourceFilterFactory.

```
<servlet>
    <servlet-name>rummy.RummyApplication</servlet-name>
    <servlet-
class>com.sun.jersey.spi.container.servlet.ServletContainer</ser
vlet-class>
```

```
    <init-param>
        <param-name>javax.ws.rs.Application</param-name>
        <param-value>rummy.RummyApplication</param-value>
    </init-param>
    <init-param>
        <param-
name>com.sun.jersey.spi.container.ResourceFilters</param-name>
        <param-
value>com.sun.jersey.api.container.filter.RolesAllowedResourceFi
lterFactory</param-value>
    </init-param>
</servlet>

<servlet-mapping>
    <servlet-name>rummy.RummyApplication</servlet-name>
    <url-pattern>/resources/*</url-pattern>
</servlet-mapping>
```

5. Apply a @Roles allowed annotation and test it.
   - Currently any authenticated user that is in the player or the admin role can list all the games in the system. The web.xml file has a security constraint for the <url-pattern>/*</url-pattern> that requires the player or admin role.
   - Because web.xml security constraints are used to restrict HTTP method and URL pattern combinations to roles, you can define many restrictions just by using a web.xml approach when creating JAX-RS endpoints. JAX-WS endpoints send everything through a POST request and, therefore, you can't always use web.xml to limit endpoint method calls to roles.
   - Open a command prompt and cd to D:\labs\student\resources.
   - Use cURL to verify that any authenticated player can list all games (use a user that is valid for you).

```
curl -v -u matt -k -H "Content-Type: application/json" -X GET
https://localhost:7002/IndianRummyWS/resources/games
```

   - You should receive a 200 OK response.
   - Open the RummyGameCollectionResource.java file.
   - Annotate the getGames method to restrict access to users in the admin role.

```
@RolesAllowed({"admin"})
@GET
public List<RummyGame> getGames(@Context UriInfo uriInfo) {
```

   - Add any required imports and save your changes.
   - Use cURL again to see if any player can list all games (use a user that is valid for you).

```
curl -v -u matt -k -H "Content-Type: application/json" -X GET
https://localhost:7002/IndianRummyWS/resources/games
```

   - You should receive a 403 Forbidden response.
6. Use a SecurityContext to programmatically access principal and role information.

- Currently there is a huge security hole in the rummy application. If you didn't spot it that is because it can be difficult to understand how all the security-constraints in the web.xml file interact. Using @RolesAllowed and SecurityContext with a minimal set of security constraints in web.xml can result in a more understandable configuration.
- Currently in web.xml you should have the following security constraint.

```
<security-constraint>
    <display-name>Indian Rummy Add Player</display-name>
    <web-resource-collection>
        <web-resource-name>Add player PUT</web-resource-name>
        <description>Navigate to resource</description>
        <url-pattern>/resources/*</url-pattern>
        <http-method>PUT</http-method>
    </web-resource-collection>
    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
</security-constraint>
```

- This constraint is missing an <auth-constraint> element and therefore any user, unauthenticated or not, can perform a PUT to any resource. The idea behind this constraint is that unauthenticated users must be able to create an account; otherwise they won't have credentials to authenticate with. Account creation can be done like this:

```
curl -v -k -H "Content-Type: application/json" -X PUT --data
"{\"userName\":\"matt\",\"password\":\"welcome1\"}"
https://localhost:7002/IndianRummyWS/resources/players/matt
```

- When creating a new user, you receive a 201 Created response.
- Repeating that same request does not create a new resource, instead it updates the existing resource. Currently, an unauthenticated person can change the name of an existing player or change their password. It is not the most secure system.
- You could attempt to split player creation and update into different URLs or HTTP methods but a better solution is to add a little code to the createOrUpdatePlayer method in the PlayersResource class.
- Open the PlayersResource.java file and locate the createOrUpdatePlayer method.
- Add a new method parameter of @Context SecurityContext sCtx.

```
public Response createOrUpdatePlayer(@Context SecurityContext
sCtx, @Context UriInfo uriInfo, @PathParam("id") String
userName, Player player) {
```

- Use SecurityContext to determine if there is an authenticated user, if that user is in the admin role, and if the calling principal's name matches the player to be changed. If there is no authenticated users an unauthorized response should be returned. If the authenticated user is not an admin or not the same user as the one being edited then return a forbidden response. Else create or update the player.

```
if (isUserPresent(userName)) {
    if (sCtx.getUserPrincipal() == null) {
        ResponseBuilder responseBuilder =
Response.status(Response.Status.UNAUTHORIZED);
```

```
        return responseBuilder.build();
    } else if(!sCtx.isUserInRole("admin") &&
!sCtx.getUserPrincipal().getName().equals(userName)) {
        ResponseBuilder responseBuilder =
Response.status(Response.Status.FORBIDDEN);
        return responseBuilder.build();
    } else {
        if (userName.equals(player.getUserName())) {
            if (player.getPassword() != null) {
                setPassword(player.getUserName(),
player.getPassword());
            }
        } else {
            String statsString = getUserDescription(userName);
            deletePlayer(userName);
            createUser(player.getUserName(),
player.getPassword(), statsString);
            joinGroup(userName, "players");
        }
        ResponseBuilder responseBuilder = Response.ok();
        return responseBuilder.build();
    }
} else {
```

- If a caller is not authenticated, a null will be returned when calling getUserPrincipal. Some implementations might return an "anonymous" principal. If they are not logged in but are still trying to update an existing user, a 401 Unauthorized response is returned.
- If they are logged in but are not the admin or not the user being updated then they are a player trying to change an account that is not theirs. Respond with a 403 Forbidden status.

(Optional) Practice 14-2: Using Additional Jersey Filters

Overview

In this practice, you configure additional Jersey filters.

Tasks

1. Open the web.xml configuration file for the IndianRummyWS project.
2. Add additional initial parameters to the Jersey Servlet that enable request and response logging.

```
<init-param>
    <param-
name>com.sun.jersey.spi.container.ContainerRequestFilters</param
-name>
    <param-
value>com.sun.jersey.api.container.filter.LoggingFilter</param-
value>
</init-param>
<init-param>
    <param-
name>com.sun.jersey.spi.container.ContainerResponseFilters</para
m-name>
    <param-
value>com.sun.jersey.api.container.filter.LoggingFilter</param-
value>
</init-param>
```

3. Save your changes and make a request to a resource in your application. You should see the request and response in the NetBeans Output window. These filters achieve effectively the same thing as the JVM -Dcom.sun.xml.ws.transport.http.HttpAdapter.dump=true system property does for JAX-WS services.
4. Open the D:\labs\student\resources\jersey\apidocs\jersey\index.html page. View the Javadocs for each filter.
5. The com.sun.jersey.api.container.filter package contains all the provided filters.
6. Apply the CsrfProtectionFilter to incoming requests.
   - When using multiple container filters, separate their class names with a semi-colon.

```
<init-param>
    <param-
name>com.sun.jersey.spi.container.ContainerRequestFilters</param
-name>
    <param-
value>com.sun.jersey.api.container.filter.LoggingFilter;com.sun.
jersey.api.container.filter.CsrfProtectionFilter</param-value>
</init-param>
```

   - Now if you attempt a PUT request, you will receive a 400 Bad Request response even if you are authenticated correctly. For example:

```
curl -v -u weblogic -k -H "Content-Type: application/json" -X
PUT --data "{\"userName\":\"matt\",\"password\":\"welcome1\"}"
https://localhost:7002/IndianRummyWS/resources/players/matt
```

- To correctly place a request you must now include the X-Requested-By header:

```
curl -v -u weblogic -k -H "X-Requested-By: curl" -H "Content-
Type: application/json" -X PUT --data
"{\"userName\":\"matt\",\"password\":\"welcome1\"}"
https://localhost:7002/IndianRummyWS/resources/players/matt
```

Practices for Lesson 15: Authenticating with OAuth and Jersey

Chapter 15

Practices for Lesson 15: Authenticating with OAuth and Jersey

Practices Overview

In these practices, you will allow a client application to access a protected resource on behalf of a resource owner by using OAuth.

Practice 15-1: Configuring OAuth Compatible HTTP Security Restrictions

Overview

In this practice, you modify the web.xml configuration file and apply @RolesAllowed annotations to make the IndianRummyWS application compatible with OAuth.

OAuth uses the Authorization HTTP header. For example, a request to obtain a temporary credential might include the following example as a header:

```
Authorization: OAuth oauth_version="1.0",
oauth_signature_method="HMAC-SHA1",
oauth_nonce="tt8JrgOT5IQvxuZ", oauth_timestamp="1363664206",
oauth_consumer_key="834218a34ca849ba87edf4d8dac74d2e",
oauth_signature="hC4NMbVtRF0tn7G92L6fjZLQEws%3D"
```

When using HTTP Basic authentication you also use the Authorization HTTP header. A request with a Basic-encoded username and password would look like this:

```
Authorization: Basic bWF0dDp3ZWxjb21lMQ==
```

If you have a web.xml configuration with an <auth-constraint> like this:

```
<security-constraint>
    <display-name>Indian Rummy Root Path</display-name>
    <web-resource-collection>
        <web-resource-name>everything</web-resource-name>
        <url-pattern>/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <description/>
        <role-name>player</role-name>
        <role-name>admin</role-name>
    </auth-constraint>
    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
</security-constraint>

<login-config>
    <auth-method>BASIC</auth-method>
</login-config>
```

it becomes difficult to use OAuth. The web.xml configuration takes precedence over anything in your resource classes or the Jersey configuration like the OAuth filter. The security constraint in the web.xml requires that the Authorization header be used for a Basic encoded username and password when authenticating your session.

The OAuth specification allows the OAuth parameter to be located in other locations (see section 3.4.1.3.1 Parameter Sources of RFC5849) but specifies that the Authorization header is the preferred location. Many OAuth tools only support OAuth parameters in the Authorization header.

You could try switching to a different form of authentication such as FORM-based authentication or a custom solution using the programmatic login support in Java EE 6 but the use of Basic authentication is the easiest for clients to support.

A simple solution is to avoid using the web.xml to enforce role restrictions. If you take the example above and just remove the <auth-constraint> element then the web container will not require the use of a Basic encoded username and password but HTTPS would still be enforced.

After removing the <auth-constraint> all unauthenticated users can reach your Jersey resources, which it turns out is what you want. If you use the @RolesAllowed annotations you can restrict access to your resource classes.

The only drawback to using @RolesAllowed instead of <auth-constraint> is that clients will receive a 403 Forbidden response instead of a 401 Unauthorized. Remember that web browsers prompt you for a username and password when receiving a 401 response but only display an error page when receiving a 403 so there is no way for a standard web browser to add an Authorization header with a BASIC encoded username and password. Tools like RESTClient and cURL can still add the BASIC encoded Authorization header and the server will use the supplied (but unprompted) credentials to authenticate a user.

If you look in the RESTClient extension, under the Authentication menu you will see a Basic Authentication item that adds a correctly encoded Authentication header for BASIC.

Assumptions

You have completed Practice 14-1: Using Java EE Roles and Principles.

Tasks

1. Open the IndianRummyWS project in NetBeans.
2. Open the web.xml configuration file (Source view).
3. Comment out or remove the existing <security-constraint> elements.
4. Add a new <security-constraint> element that requires the use of HTTPS for all URLs but does not require any authenticated roles.

```
<security-constraint>
    <display-name>Indian Rummy Application</display-name>
    <web-resource-collection>
        <web-resource-name>everything</web-resource-name>
        <url-pattern>/*</url-pattern>
    </web-resource-collection>
    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
</security-constraint>
```

5. Using either cURL, a web browser, or the RESTClient extension – request the https://localhost:7002/IndianRummyWS/resources/ URL with no authentication credentials. You should receive a 200 OK response.
6. Open the IndianRummyRootResource.java file and add a `@RolesAllowed({"player","admin"})` annotation at the class level.
7. Add any required imports and save your changes.

8. Using either cURL, a web browser, or the RESTClient extension – request the https://localhost:7002/IndianRummyWS/resources/ URL with no authentication credentials. You should receive a 403 Forbidden response.
9. Using either cURL or the RESTClient extension use a valid username and password to access the https://localhost:7002/IndianRummyWS/resources/ URL. You should receive a 200 OK response.

    Note: If you used RESTClient you can now open another tab in FireFox and reach the protected resource. Because the RESTClient shares cookies with all other FireFox tabs, any tab going to the same location will take part in the authenticated session.
10. Because the @RolesAllowed annotation was placed on the root resource, all sub resources are also protected.
11. Open the RummyGameCollectionResource.java file.
12. Add the @RolesAllowed({"admin"}) annotation to the deleteGame method.
13. Save your changes.
14. Use a non-admin user and verify that you receive a 403 Forbidden response when trying to send a DELETE request to https://localhost:7002/IndianRummyWS/resources/games.

Practice 15-2: Configuring Jersey OAuth Components

Overview

In this practice you configure the Jersey-provided OAuth components. By themselves, the provided Jersey components are not a functional OAuth implementation. In the next practice you will add the application-supplied components such as client registration and resource owner authorization. Additionally, the DefaultOAuthProvider class used in these exercises is a sample implementation of the OAuthProvider interface. The DefaultOAuthProvider class stores all credentials in memory. In a production environment you would supply your own implementation of the OAuthProvider interface.

Because the DefaultOAuthProvider class stores everything in memory, every time you redeploy the application you will lose all stored credentials. Additionally, every time you save a file in NetBeans the project will automatically be re-deployed. Saving a file means losing all credentials.

Assumptions

You have completed Practice 15-1: Configuring OAuth Compatible HTTP Security Restrictions.

Tasks

1. Open the IndianRummyWS project in NetBeans.
2. Create a new `rummy.oauth` Java package.
3. The first step is enabling the injection of an OAuthProvider instance. An OAuthProvider is used on the server-side for all credential management. In your resource class you will obtain an OAuthProvider like this:

```
private @Context OAuthProvider oauth;
```

4. JAX-RS uses its own form of dependency injection in the form of the @Context annotation. The JAX-RS specification outlines a small number of types that may be injected using the @Context annotation. Using Jersey-specific enhancements you can allow other class types to be injected.
5. Create an InjectableProvider that supplies a DefaultOAuthProvider instance when an OAuthProvider should be injected.
   - Create an OAuthContextResolver class in the rummy.oauth package.
   - Complete the OAuthContextResolver as shown:

```
import
com.sun.jersey.oauth.server.api.providers.DefaultOAuthProvider;
import com.sun.jersey.oauth.server.spi.OAuthProvider;
import
com.sun.jersey.spi.inject.SingletonTypeInjectableProvider;
import javax.ws.rs.core.Context;
import javax.ws.rs.ext.Provider;

@Provider
public class OAuthContextResolver extends
SingletonTypeInjectableProvider<Context, OAuthProvider>{

    public OAuthContextResolver() {
```

```
        super(OAuthProvider.class,
                new DefaultOAuthProvider());
    }

}
```

6. Register the OAuthContextResolver class in the RummyApplication class.
   - Open the RummyApplication.java file.
   - In the getSingletons method, add a new OAuthContextResolver instance to the set of singletons.

```
set.add(new OAuthContextResolver());
```

   -
7. Register the AccessTokenRequest and RequestTokenRequest classes as resource classes.
8. In the RummApplication.java file find the getClasses method.
9. Add AccessTokenRequest and RequestTokenRequest instances to the set of resource classes returned by the getClasses method.

```
s.add(com.sun.jersey.oauth.server.api.resources.AccessTokenReque
st.class);
s.add(com.sun.jersey.oauth.server.api.resources.RequestTokenRequ
est.class);
```

10. Open the web.xml configuration file.
11. Add the OAuthServerFilter container request filter.
    - You should have an existing com.sun.jersey.spi.container.ContainerRequestFilters init-param with a value. To add a second value, separate the class names with a semi-colon.

```
<init-param>
    <param-
name>com.sun.jersey.spi.container.ContainerRequestFilters</param
-name>
    <param-
value>com.sun.jersey.api.container.filter.LoggingFilter;com.sun.
jersey.oauth.server.api.OAuthServerFilter</param-value>
</init-param>
```

12. Configure the OAuthServerFilter to ignore requests to URLs for the request token and access token resources.
    - The OAuthServerFilter throws a WebApplicationException when the OAuth credentials supplied in the Authorization header do not resolve to a valid security context. While obtaining temporary credentials and token credentials the filter would throw the exception because the OAuth parameters are not complete yet. The filter should ignore the resource URLs used to obtain temporary credentials and token credentials.
    - Add another init-param to the Application sub-class servlet.

```
<init-param>
    <param-
name>com.sun.jersey.config.property.oauth.ignorePathPattern</par
am-name>
```

```
    <param-value>requestToken|accessToken</param-value>
</init-param>
```

Note: the OAuthServerFilter automatically ignores requests with no Authorization header or an Authorization header that contains no OAuth parameters.

Practice 15-3: Create Registration and Approval Resources

Overview

In this practice, you create the resources needed to enable clients to register with your OAuth server. A client in this case is the piece of software that will use OAuth to connect to your service. You will also create the resource method to enable resource owners to authorize temporary credentials.

In this practice you will output HTML from a JAX-RS resource class. While producing some small HTML output might be acceptable in production applications you would typically avoid producing a large number of HTML interfaces from within a JAX-RS resource class. To do so is similar to only using Servlets in a web application when JSPs are available.

We will not be using it because it is beyond the scope of this class but if you want to use JAX-RS resource classes as controllers in an MVC web application with JSP views, there is a com.sun.jersey.api.view.Viewable class that is the Jersey equivalent of a RequestDispatcher for MVC design.

Assumptions

You have completed Practice 15-2: Configuring Jersey OAuth Components.

Tasks

1. Create a new OAuthRootResource class in the rummy.oauth package.
2. Annotate the OAuthRootResource class.

```
@RolesAllowed({"player", "admin"})
@Path("/oauth")
@Produces({MediaType.TEXT_HTML})
public class OAuthRootResource {
```

3. Add injected fields.

```
private @Context
UriInfo uriInfo;
private @Context
SecurityContext secCtx;
private @Context OAuthProvider oauth;
private UriBuilder uriBuilder;
```

4. Initialize the UriBuilder in a @PostConstruct method.

```
@PostConstruct
private void init() {
    uriBuilder = uriInfo.getBaseUriBuilder().path("oauth");
}
```

5. There will be two things that can be done using this resource. Register a new client and view a list of clients registered under your login. The /resources/oauth GET response should be a web page with links to the two activities. Add the getPage method.

```
@GET
public String getPage() {
    StringBuilder sb = new StringBuilder();
```

```
    sb.append("<html>");
    sb.append("<head><title>OAuth</title></head>");
    sb.append("<body>");
    sb.append("<a href='");
    sb.append(uriBuilder.clone().path("register").build());
    sb.append("'>Register</a>");
    sb.append("<br/>");
    sb.append("<a href='");
    sb.append(uriBuilder.clone().path("list").build());
    sb.append("'>List</a>");
    sb.append("</body>");
    sb.append("</html>");
    return sb.toString();
}
```

6. The registration page is simply a form that asked for the name of the client application to register. Add the getRegisterPage method.

```
@Path("register")
@GET
public String getRegisterPage() {
    StringBuilder sb = new StringBuilder();
    sb.append("<html>");
    sb.append("<head><title>OAuth</title></head>");
    sb.append("<body>");
    sb.append("<form action='");
    sb.append(uriBuilder.clone().path("register").build());
    sb.append("' method='POST'>");
    sb.append("<label for='name-input'>Application
Name:</label><input id='name-input' type='text' name='name'/>");
    sb.append("<input type='submit' value='Register'/>");
    sb.append("</form>");
    sb.append("</body>");
    sb.append("</html>");
    return sb.toString();
}
```

7. There should be a postForm method that receives the form submission from the form produced by the getRegisterPage method.

```
@Path("register")
@POST
public String postForm(Form form) {
    String developer = secCtx.getUserPrincipal().getName();
    Consumer client = ((DefaultOAuthProvider)
oauth).registerConsumer(developer, form);
```

```
    StringBuilder sb = new StringBuilder();
    sb.append("<html>");
    sb.append("<head><title>OAuth</title></head>");
    sb.append("<body>");
    sb.append("<h1>Client Credentials</h1>");
    sb.append("Name: ");
    sb.append(client.getAttributes().getFirst("name"));
    sb.append("<br/>");
    sb.append("Key: ");
    sb.append(client.getKey());
    sb.append("<br/>");
    sb.append("Secret: ");
    sb.append(client.getSecret());
    sb.append("<br/>");
    sb.append("<a href='");
    sb.append(uriBuilder.clone().build());
    sb.append("'>Back</a>");
    sb.append("</body>");
    sb.append("</html>");
    return sb.toString();
}
```

8. Add a getListPage method that will produce a list of all client credentials for the current user.

```
@Path("list")
@GET
public String getListPage() {
    StringBuilder sb = new StringBuilder();
    sb.append("<html>");
    sb.append("<head><title>OAuth</title></head>");
    sb.append("<body>");
    String developer = secCtx.getUserPrincipal().getName();
    Set<Consumer> clientCreds = ((DefaultOAuthProvider)
oauth).getConsumers(developer);
    if (clientCreds.size() > 0) {
        sb.append("<h1>Client Credentials</h1>");
        for (Consumer client : clientCreds) {
            sb.append("Name: ");
            sb.append(client.getAttributes().getFirst("name"));
            sb.append("<br/>");
            sb.append("Key: ");
            sb.append(client.getKey());
            sb.append("<br/>");
```

```
            sb.append("Secret: ");
            sb.append(client.getSecret());
            sb.append("<br/><br/>");
        }
    } else {
        sb.append("You have no client credentials under the name
");
        sb.append(developer);
    }
    sb.append("<br/>");
    sb.append("<a href='");
    sb.append(uriBuilder.clone().build());
    sb.append("'>Back</a>");
    sb.append("</body>");
    sb.append("</html>");
    return sb.toString();
}
```

9. Add any required imports.
   - When asked you should select import statements for classes under the javax.ws.rs, com.sun.jersey and javax.annotation paths.
10. Add the OAuthRootResource class to the RummyApplication class.
    - Open the RummyApplication class.
    - Locate the getClasses method.
    - Add the OAuthRootResource class to the set returned by the getClasses method.

      ```
      s.add(OAuthRootResource.class);
      ```

    - Add any required imports and save your changes.
11. Deploy the IndianRummyWS project.
12. Request client credentials and temporary credentials.
    - If any FireFox or RESTClient windows are open, close them all.
    - Start FireFox and launch the RESTClient extension.
    - Perform a GET request to https://localhost:7002/IndianRummyWS/resources/oauth.
    - You should receive a 403 Forbidden response.
    - Add a Basic Authorization header. You can use weblogic/welcome1 as the username/password.
    - Send the request again; you should receive a 200 OK response.
    - In the same browser window, open a new tab and visit https://localhost:7002/IndianRummyWS/resources/oauth.
    - Because you established an authenticated session in the RESTClient tab, all FireFox tabs will share that session.
    - Click the List hyperlink; you should have no client credentials.
    - Use the Register hyperlink to register a new application. You can use any name for the client application.
    - Upon submitting the registration form you should be presented with output similar to:

```
Name: MyOAuthClient
Key: 6c7f1e74007743bd8528232b9e3f7aab
Secret: c5f8bcc439a64f46a46e712b08391784
```

- Copy and paste this information to a text document.
- Return to the RESTClient tab.
- Remove all existing headers.
- Using the settings menu, choose List request headers in table. If the option is not available you are already displaying the headers in table format. The table format will allow you to see the complete value of any header.
- Using the setting menu, choose Switch to fixed page layout. If the option is not available you are already using a fixed page layout. Currently there is a bug in RESTClient that prevents one of the OAuth dialog boxes from appearing when using percentage page layout.
- Request temporary credentials (also known as a request token and secret).
  - Change the URL in the RESTClient extension to https://localhost:7002/IndianRummyWS/resources/requestToken.
  - Change the method POST.
  - Add a Content-Type header with a value of application/x-www-form-urlencoded.
  - Open the Authentication menu and select OAuth.
  - Enter your client credentials (consumer key and secret).
  - Leave the token credentials (access token and secret) blank (they will show grayed out example values).
  - Click the Insert button.
  - In the Notice dialog, answer "Yes, please" to the prompt about automatically updating the OAuth signature.
  - Observe the generated Authorization header value.
  - You should receive a 200 OK response after clicking SEND.
  - View the Response Body (Raw) tab. You should see text similar to:

```
oauth_callback_confirmed=true&oauth_token=6e8112a8987b4d66a6f296
523bebcdce&oauth_token_secret=5654ef8e4ac2496688d105f4d2f526cc
```

  - Copy this information to the same text document that contained the client (consumer) credentials.
  - If you receive a 403 Forbidden response, it is likely that your authenticated session timed out. Simply repeat task 10 without interruption.

13. At this point the client (application) is done talking to the server. It has its client (consumer key and secret) and temporary (request key and secret) credentials.
14. Close ALL FireFox and RESTClient windows. The purpose of this step is to "logout" by discarding your session cookies.
15. The client application now asks the resource owner to log in to the OAuth provider and approve the request. This can either take the form of a redirect or a prompt to visit the authorization page on the server. The authorization page is a developer-supplied resource.
16. Create the authorization form.
    - Open the AuthRootResource.java file.

- Add a getAuthorizePage method that displays the authorization form. Note that the query parameter named oauth_token passes in the temporary (request) token key. This value is the one returned at the end of task 12.

```
@Path("authorize")
@GET
public String getAuthorizePage(@QueryParam("oauth_token") String
token) {
    StringBuilder sb = new StringBuilder();
    sb.append("<html>");
    sb.append("<head><title>OAuth</title></head>");
    sb.append("<body>");
    String owner = secCtx.getUserPrincipal().getName();
    if (token == null) {
        sb.append("Missing oauth_token parameter");
    } else if (oauth.getRequestToken(token) == null ||
oauth.getRequestToken(token).getConsumer() == null) {
        sb.append("Those temporary credentials do not exist");
    } else {
        sb.append("<h1>Resource Owner Authorization</h1>");
        sb.append("<form action='");
        sb.append(uriBuilder.clone().path("authorize").build());
        sb.append("' method='POST'>");
        sb.append("<input type='hidden' name='token' value='");
        sb.append(token);
        sb.append("'/>");
        sb.append("<label for='auth-select'>Approve
Request:</label>");
        sb.append("<select id='auth-select' name='auth-
select'>");
        sb.append("<option selected value='yes'>Yes</option>");
        sb.append("<option value='no'>No</option>");
        sb.append("</select>");
        sb.append("<input type='submit' value='Submit'/>");
        sb.append("</form>");
    }
    sb.append("</body>");
    sb.append("</html>");
    return sb.toString();
}
```

17. Create the handleAuthorizeForm method that handles the form submission for the form created in the previous task. In particular, notice the third argument to the authorizeToken method that is a set of role names. The result of this method is an an oauth_verifier value.

```
@Path("authorize")
@POST
```

```
public String handleAuthorizeForm(@FormParam("token") String
token, @FormParam("auth-select") String approvedStatus) {
    DefaultOAuthProvider.Token tempCred =
((DefaultOAuthProvider) oauth).getRequestToken(token);
    StringBuilder sb = new StringBuilder();
    sb.append("<html>");
    sb.append("<head><title>OAuth</title></head>");
    sb.append("<body>");
    String owner = secCtx.getUserPrincipal().getName();
    if (tempCred == null || tempCred.getConsumer() == null) {
        sb.append("Those temporary credentials do not exist");
    } else if (approvedStatus.equalsIgnoreCase("yes")) {
        Set<String> roles = new HashSet<>();
        roles.add("player");
        String verifier = ((DefaultOAuthProvider)
oauth).authorizeToken(tempCred, secCtx.getUserPrincipal(),
roles);
        sb.append("Your verification code is: ");
        sb.append(verifier);
    } else {
        sb.append("You denied the request");
    }
    sb.append("</body>");
    sb.append("</html>");
    return sb.toString();
}
```

Add any required imports and save all your changes.

Saving all your changes results in the loss of all credentials.

All the components are complete; all that is left is to work through the entire flow.

Practice 15-4: Executing the OAuth Flow

Overview

In this practice, you use all the resources you have created to login as a user that has approved an OAuth temporary token.

Assumptions

You have completed Practice 15-3: Create Registration and Approval Resources.

Tasks

1. Request client credentials and temporary credentials.
   - If any FireFox or RESTClient windows are open, close them all.
   - Start FireFox and launch the RESTClient extension.
   - Perform a GET request to https://localhost:7002/IndianRummyWS/resources/oauth.
   - You should receive a 403 Forbidden response.
   - Add a Basic Authorization header. You can use weblogic/welcome1 as the username/password.
   - Send the request again; you should receive a 200 OK response.
   - In the same browser window, open a new tab and visit https://localhost:7002/IndianRummyWS/resources/oauth.
   - Because you established an authenticated session in the RESTClient tab, all FireFox tabs will share that session.
   - Click the List hyperlink; you should have no client credentials.
   - Use the Register hyperlink to register a new application. You can use any name for the client application.
   - Upon submitting the registration form you should be presented with output similar to:

     ```
     Name: MyOAuthClient
     Key: 0380777a22cf433e988f5e9424a813b7
     Secret: 019182e2fad5415991ea19c7637cd6bf
     ```

   - Copy and paste this information to a text document.
   - Return to the RESTClient tab.
   - Remove all existing headers.
   - Using the settings menu, choose List request headers in table. If the option is not available you are already displaying the headers in table format. The table format will allow you to see the complete value of any header.
   - Using the setting menu, choose Switch to fixed page layout. If the option is not available, you are already using a fixed page layout. Currently there is a bug in RESTClient that prevents one of the OAuth dialog boxes from appearing when using percentage page layout.
   - Request temporary credentials (also known as a request token and secret).
     - Change the URL in the RESTClient extension to https://localhost:7002/IndianRummyWS/resources/requestToken.
     - Change the method POST.
     - Add a Content-Type header with a value of application/x-www-form-urlencoded.
     - Open the Authentication menu and select OAuth.

- Enter your client credentials (consumer key and secret).
- Leave the token credentials (access token and secret) blank (they will show grayed out example values).
- Click the Insert button.
- In the Notice dialog, answer "Yes, please" to the prompt about automatically updating the OAuth signature.
- Observe the generated Authorization header value.
- You should receive a 200 OK response after clicking SEND.
- View the Response Body (Raw) tab. You should see text similar to:

```
oauth_callback_confirmed=true&oauth_token=a7e059081c924486a1443c
d2e339cae1&oauth_token_secret=ac59b34f77764e25ab584b3f3c89e37b
```

- Copy this information to the same text document that contained the client (consumer) credentials.
- If you receive a 403 Forbidden response, it is likely that your authenticated session timed out. Simply repeat task 1 without interruption.

2. At this point the client (application) is done talking to the server. It has its client (consumer key and secret) and temporary (request key and secret) credentials.
3. Close ALL FireFox and RESTClient windows. The purpose of this step is to "logout" by discarding your session cookies.
4. Launch FireFox and start the RESTClient extension.
5. Log in to the IndianRummyWS application using a different account than the one that was used to generate the client credentials.
   - In the RESTClient extension, change the URL to https://localhost:7002/IndianRummyWS/resources/oauth.
   - Add a Basic Authentication header using a valid player account such as matt/welcome1.
   - You should receive a 200 OK response after clicking SEND.
   - Now that "matt" has an authenticated session, look to see if he has any client credentials by opening https://localhost:7002/IndianRummyWS/resources/oauth in another FireFox tab and clicking the List hyperlink.
   - You do not need to register any new client credentials.
6. Use FireFox to approve a temporary credential (request token).
   - Use FireFox to visit https://localhost:7002/IndianRummyWS/resources/oauth/authorize?oauth_token=a7e059081c924486a1443cd2e339cae1 (where the value after oauth_token?= is the value in the text you copied at the end of task 1).
   - You should be presented with a form asking you to approve an authorization request.
   - Click the Submit button to approve the request.
   - Copy the verification code to the text document with your other information.

```
Your verification code is: 70c0ce3f8e0d48159ffe42c5fdb872a9
```

7. Return to the RESTClient tab.
8. Remove all existing headers.
9. Finish by verifying the request and generating the token credential (access token).
   - Change the URL in the RESTClient extension to https://localhost:7002/IndianRummyWS/resources/accessToken?oauth_verifier=VERIFICATION_CODE_HERE (there is an underscore in oauth_verifier)

- Change the method to POST.
- Add a Content-Type header with a value of application/x-www-form-urlencoded.
- Open the Authentication menu and select OAuth.
- Enter your client credentials (consumer key and secret).
- Fill out the token credentials (access token and secret) using the oauth_token and oauth_token_secret values obtained at the end of task 1.
- Click the Insert button.
- In the Notice dialog, answer "Yes, please" to the prompt about automatically updating the OAuth signature.
- Observe the generated Authorization header value.
- You should receive a 200 OK response after clicking SEND.
- View the Response Body (Raw) tab. You should see text similar to:

```
oauth_token=bfa4894a3e1a4b6eae29c0b8c0d23e3e&oauth_token_secret=
b0c6c927bfc9474c981963efaae45567
```

- Copy this information to the same text document that contained the client (consumer) credentials. These are your token credentials (access token and secret).

10. Use the RESTClient extension to test the token credentials.
    - Currently you are logged in as a user that has just approved OAuth credentials.
    - Close all FireFox and RESTClient windows to log out.
    - Start FireFox and launch the RESTClient extension.
    - Using the RESTClient extension, try to view the player resource of the player that just approved the OAuth request.

      ```
      https://localhost:7002/IndianRummyWS/resources/players/matt
      ```

    - You should receive a 403 Forbidden response.
    - Using the Authentication > OAuth menu item, and add the client credentials (consumer key & secret) and token credentials (access token & secret).
      - The client credentials or consumer key and secret were the first values obtained in task 1. In this example:
      - Key: 0380777a22cf433e988f5e9424a813b7
      - Secret: 019182e2fad5415991ea19c7637cd6bf
      - The token credentials or access token and secret were the values obtained at the end of task 9. In this example:
      - Key: bfa4894a3e1a4b6eae29c0b8c0d23e3e
      - Secret: b0c6c927bfc9474c981963efaae45567

      Now request for the https://localhost:7002/IndianRummyWS/resources/players/matt resource again. The result should be a 200 OK response with a method body containing matt's details.
11. Congratulations, you have set up OAuth on your server. There are a few remaining tasks for a production environment:
    - Replace the DefaultOAuthProvider with one that has persistence. Jersey is an open-source project, so you may wish to explore the source code for DefaultOAuthProvider used in this practice.
    - Implement key management. Resource owners should have a way to view a list of their authorized keys and revoke them.

- Create a Jersey client that uses AOuth. There is a client-side filter you can use with the Jersey client API.
- Add additional roles. Right now the agent using the OAuth credentials effectively has the same permissions as a regular player. You may want to allow authenticated agents to only perform a sub-set of activities that the resource owner can perform. Attaching different roles when calling authorizeToken() combined with both @RolesAllowed type declarative security and programmatic security provides a versatile security configuration.

WebLogic Server

Chapter 16

Appendix A

Practices Overview

In these practices, you configure your environment to develop Java EE 6 applications.

Practice A-1: Installing WebLogic Server

Overview

In this practice, you install the Oracle WebLogic Server 12*c* (12.1.1) Zip Distribution.

Assumptions

JDK 7 is installed.

NetBeans 7.2 EE is installed.

The Oracle WebLogic Server 12*c* (12.1.1) Zip Distribution has been downloaded.

Tasks

1. Create the `D:\weblogic\wls` directory, which contains the WebLogic application server.
2. Unzip `D:\weblogic\wls1211_dev.zip` into `D:\weblogic\wls`.
3. Open a command prompt and run the following commands:

   Note: In the following sequence, the `JAVA_HOME` value cannot contain any spaces. You can use the `dir /X` command to discover a directory's short name.

```
D:
cd D:\weblogic\wls
set JAVA_HOME=D:\PROGRA~2\Java\jdk1.7.0_17
set MW_HOME=D:\weblogic\wls
set JAVA_VENDOR=Sun
configure.cmd
mkdir D:\weblogic\domain
cd D:\weblogic\domain
%MW_HOME%\wlserver\server\bin\setWLSEnv.cmd
%JAVA_HOME%\bin\java.exe %JAVA_OPTIONS% -Xmx1024m
          -XX:MaxPermSize=128m weblogic.Server
```

   Note: For your convenience, there is a batch file, `D:\labs\resources\weblogic_install.bat`, that can be edited to perform the steps above.
4. Answer the installation questions:
   - Create a default configuration and boot: **y**
   - Enter username to boot WebLogic server: **weblogic**
   - Enter password to boot WebLogic server: **welcome1**
5. Open the administration console.
   - Open a web browser and visit http://localhost:7001/console.
   - Log in with the username and password specified in the previous step.
6. Shut down WebLogic.
   - In the command prompt window, which was used to install WebLogic, press Ctrl + C.

Notes for Installing with a 64-bit JDK

If you get the following error when trying to create a domain from the command line:

```
<Dec 13, 2011 9:27:37 AM CST> <Error> <Security> <BEA-090783>
<Server is Running in Development Mode and Native
Library(terminalio) to read the password securely from
commandline is not found.>
```

Then, you are using a 64-bit JDK and need to make the following changes (or switch to a 32-bit JDK): Edit `%MW_HOME%\wlserver\common\bin\commEnv.cmd`.

- Find the following variables and change their values as shown.

```
set JAVA_USE_64BIT=true
set WL_USE_X86DLL=false
set WL_USE_IA64DLL=false
set WL_USE_AMD64DLL=true
```

Practice A-2: Generating Security Certificates and Keys

Overview

In this practice, you create keys and certificates for a certificate authority (CA), a server, and a client.

In this practice you will use your own self-signed certificate authority and therefore you will still get browser warnings. If you eliminate the steps related to creating your own CA, then these steps are comparable to purchasing a certificate that is signed by a trusted CA.

The keys generated here are stronger than the demo keys that ship with WebLogic Server. If you configure WebLogic Server to use these keys then Internet Explorer will be able to connect to WebLogic Server using HTTPS after accepting the warning about not trusting the identity of the certificate.

For more information about keytool, see http://docs.oracle.com/javase/7/docs/technotes/tools/windows/keytool.html.

Tasks

1. Start a command prompt.
   - Press the Windows Start button, go to Run, and key in cmd.exe and press Enter.
   - Switch to the D:\ drive if not already there.

```
D:
```

2. Make a directory to hold keystores:

```
mkdir D:\labs\student\keystores
```

3. Change into the keystores directory:

```
cd D:\labs\student\keystores
```

4. Create the certificate authority.
   - Generate the CA keys.

```
keytool -genkeypair -keyalg RSA -keysize 2048 -dname
"CN=ca,OU=OracleUniversity,O=Oracle,C=US" -keystore ca.jks -
alias ca -ext bc:c
```

   - Export the CA cert.

```
keytool -exportcert -rfc -alias ca -keystore ca.jks -file ca.pem
```

   - Copy the default cacerts keystore used by Java.

```
copy D:\PROGRA~2\Java\jre7\lib\security\cacerts .
```

   - Import your new CA cert into the copy of the cacerts keystore.

```
keytool -importcert -noprompt -storepass changeit -alias ca -
keystore cacerts -file ca.pem
```

   - Import all the existing demo CA certs into the copy of the cacerts keystore.

```
keytool -importkeystore -srckeystore
D:\weblogic\wls\wlserver\server\lib\DemoTrust.jks -destkeystore
cacerts -srcstorepass DemoTrustKeyStorePassPhrase -deststorepass
changeit
```

5. Generate the server HTTPS keys and cert.
   - Generate the server keys.

```
keytool -genkeypair -keyalg RSA -keysize 2048 -dname
"CN=localhost,OU=OracleUniversity,O=Oracle,C=US" -keystore
server.jks -alias server
```

- Generate a CSR for the server.

```
keytool -certreq -alias server -keystore server.jks -file
server.csr
```

- Use the CA to sign the server cert.

```
keytool -gencert -alias ca -rfc -keystore ca.jks -infile
server.csr -outfile server.pem
```

- Add the CA cert to the server.

```
keytool -importcert -alias ca -keystore server.jks -file ca.pem
```

- Add the server's signed-cert back into its keystore.

```
keytool -importcert -alias server -keystore server.jks -file
server.pem
```

6. Generate the client keys and cert.
   - Generate the client keys.

```
keytool -genkeypair -keyalg RSA -keysize 2048 -dname
"CN=client,OU=OracleUniversity,O=Oracle,C=US" -keystore
client.jks -alias client
```

   - Generate a CSR for the client.

```
keytool -certreq -alias client -keystore client.jks -file
client.csr
```

   - Use the CA to sign the client cert.

```
keytool -gencert -alias ca -rfc -keystore ca.jks -infile
client.csr -outfile client.pem
```

   - Add the CA cert to the client.

```
keytool -importcert -alias ca -keystore client.jks -file ca.pem
```

   - Add the client's signed-cert back into his keystore.

```
keytool -importcert -alias client -keystore client.jks -file
client.pem
```

7. Close the command box.

Practice A-3: Specifying SSL Keys in WebLogic Server

Overview

In this practice, you modify WebLogic Server to use the keys generated in the previous practice. First you will configure WebLogic Server to use the keys and certificates for HTTPS connections. Second you will configure Web Service Security keys.

For more detailed information regarding the tasks performed in this practice view the Administration Console Online Help http://docs.oracle.com/cd/E24329_01/apirefs.1211/e24401/core/index.html and Oracle Fusion Middleware Securing WebLogic Web Services for Oracle WebLogic Server website http://docs.oracle.com/cd/E12839_01/web.1111/e13713/toc.htm.

These steps are based on configWss.py
http://docs.oracle.com/cd/E24329_01/web.1211/e24488/message.htm#i271996

Assumptions

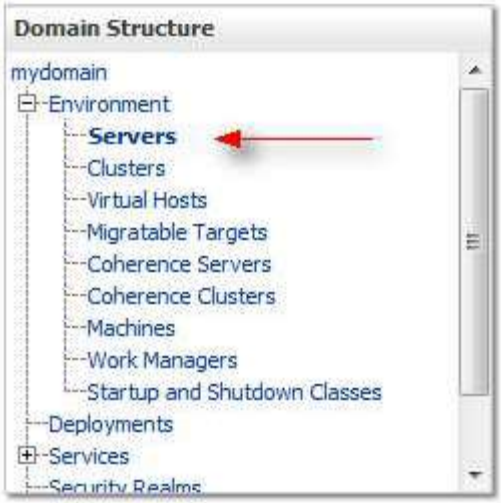
You have completed Practice 14-1: Generating Security Certificates and Keys.

Tasks

1. Using NetBeans, start WebLogic Server if it is not already started.
2. Open a web browser and visit the Administration Console.

```
http://localhost:7001/console/
```

3. Open the configuration for the myserver ketstores.
   - Under Domain Structure (on the left) open mydomain > Environment > Servers.
   - Click myserver(admin).



   - Select Configuration > Keystores.

4. Click the Change button and choose Custom Identity and Custom Trust.
5. Click Save.

6. Enter the following values:
   - Identity
   - Custom Identity Keystore: D:\labs\student\keystores\server.jks
   - Custom Identity Keystore Type: jks
   - Custom Identity Keystore Passphrase: welcome1
   - Confirm Custom Identity Keystore Passphrase: welcome1
   - Trust
   - Custom Trust Keystore: D:\labs\student\keystores\cacerts
   - Custom Trust Keystore Type: jks
   - Custom Trust Keystore Passphrase: changeit
   - Confirm Custom Trust Keystore Passphrase: changeit
7. Click Save.
8. Open the SSL tab that is next to the Keystores tab.
9. Set the value for Private Key Alias to server.
10. Enter and confirm that Private Key Passphrase: welcome1
11. Click Save.
12. Open the WebLogic Administration console in every web browser using HTTPS.

```
https://localhost:7002/console/
```

    - Whereas you will still have to confirm that you would like to continue because the CA used to sign the server's certificate is not trusted, you will now be able to make HTTPS connections using all web browsers.
    - The newly generated keys are stronger than the demo keys (512 bit vs 2048 bit). Some WS-Security features will not work with the demo keys.
    - Don't forget to confirm the security exception in FireFox or else the RESTClient extension will not be able to make HTTPS connections to your server. Remember that some URLs have a security constraint that redirect to HTTPs. These redirects will cause RESTClient to fail to load any responses unless the certificate exception has been approved.
13. Open the Web Service Security tab for the domain.
    - Click mydomin in the Domain Structure box.
    - Select the Web Service Security tab.

14. Enable an authentication provider that allows WebLogic Server to determine an identity by using the SSL certificate.
    - In the Domain Structure box, click mydomain > Security Realms.
    - Click myrealm.
    - Open the Providers > Authentication tab.
    - Click DefaultIdentityAsserter.
    - On the Configuration > Common tab, add X.509 to the Chosen column for Active Types.
    - On the Configuration > Provider Specific tab, change the "Default User Name Mapper Attribute Type" to CN and select the "Use Default User Name Mapper" box.
15. Create the default web service security configuration.
    - Create the default_wss Web Service Security configuration.
    - In the Domain Structure box, click mydomain.
    - Open the Web Service Security tab.
    - Click the New button to create a new configuration named "default_wss".
    - Click the default_wss configuration you just created.

      Note: The name must not vary; if it does then each web service must identify the security configuration by name using the @weblogic.jws.security.WssConfiguration annotation.
16. Create the default_dk_cp Credential Provider.
    - Open the Credential Provider tab.
    - Click the New button to create a new Credential Provider.
    - Set the values:
    - Name: default_dk_cp
    - Class Name: weblogic.wsee.security.wssc.v200502.dk.DKCredentialProvider
    - Token Type: dk

- Click Finish.
- Click the default_dk_cp provider you just created.
- Press the New button to add a Credential Provider Property.
- Leave the Is Encrypted box unselected and click Next.
- Enter the values:
  - Name: Label
  - Value: WS-SecureConversationWS-SecureConversation
- Click Finish.
- Click the New button to add a Credential Provider Property.
- Leave the Is Encrypted check box unselected and click Next.
- Enter the values:
  - Name: Length
  - Value: 16
- Click Finish.

17. Open the default_wss Web Service Security configuration again.
    - In the Domain Structure box, click mydomain.
    - Open the Web Service Security tab.
    - Click the default_wss configuration.
18. Create the default_x509_cp Credential Provider.
    - Open the Credential Provider tab.
    - Click the New button to create a new Credential Provider.
    - Set the values:
      - Name: default_dk_cp
      - Class Name: weblogic.wsee.security.bst.ServerBSTCredentialProvider
      - Token Type: x509
    - Click Finish.
19. Configure the Confidentiality KeyStore.
    - Click the default_x509_cp credential provider you just created.
    - Click the New button to add a Credential Provider Property.
    - Leave the Is Encrypted check box unselected and click Next.
    - Enter the values:
      - Name: ConfidentialityKeyStore
      - Value: D:\labs\student\keystores\server.jks
    - Click Finish.
    - Click the New button to add a Credential Provider Property.
    - Select the Is Encrypted check box and click Next.
    - Enter the values:
      - Name: ConfidentialityKeyStorePassword
      - Value: welcome1
    - Click Finish.
    - Click the New button to add a Credential Provider Property.

- Leave the Is Encrypted check box unselected and click Next.
- Enter the values:
  - Name: ConfidentialityKeyAlias
  - Value: server
- Click Finish.
- Click the New button to add a Credential Provider Property.
- Select the Is Encrypted check box and click Next.
- Enter the values:
  - Name: ConfidentialityKeyPassword
  - Value: welcome1
- Click Finish.

20. Configure the Identity KeyStore.
    - Click the New button to add a Credential Provider Property.
    - Leave the Is Encrypted box unselected and click Next.
    - Enter the values:
      - Name: IntegrityKeyStore
      - Value: D:\labs\student\keystores\server.jks
    - Click Finish.
    - Click the New button to add a Credential Provider Property.
    - Select the Is Encrypted check box and click Next.
    - Enter the values:
      - Name: IntegrityKeyStorePassword
      - Value: welcome1
    - Click Finish.
    - Click the New button to add a Credential Provider Property.
    - Leave the Is Encrypted check box unselected and click Next.
    - Enter the values:
      - Name: IntegrityKeyAlias
      - Value: server
    - Click Finish.
    - Click the New button to add a Credential Provider Property.
    - Select the Is Encrypted check box and click Next.
    - Enter the values:
      - Name: IntegrityKeyPassword
      - Value: welcome1
    - Click Finish.

      Note: In a production environment, it is recommended that you use separate keys for integrity and confidentiality.
21. Open the default_wss Web Service Security configuration again.
    - In the Domain Structure box, click mydomain.
    - Open the Web Service Security tab.

- Click the default_wss configuration.

22. Create the default_x509_handler Token Handler.
    - Open the Token Handler tab.
    - Click the New button to create a new Token Handler.
    - Set the values:
    - Name: default_x509_handler
    - Class Name: weblogic.xml.crypto.wss.BinarySecurityTokenHandler
    - Token Type: x509
    - Click Finish.
    - Click the default_x509_handler you just created.
    - Click the New button to add a Token Handler Property.
    - Leave the Is Encrypted box unchecked and press Next.
    - Enter the values:
    - Name: UseX509ForIdentity
    - Value: false
    - Click Finish.
23. Use the Services tab in NetBeans to restart Oracle WebLogic Server.
    -