Full Stack Web Development for Liberty Mutual

Lab Exercises

Eclipse Neon/Spring 4/Spring Boot 1.3.1/Tomcat 8 Version

Version 1.0.3 October 3, 2017



Copyright © Descriptor Systems, 2017
All Rights Reserved.

This document may not be copied or reproduced in any form without prior written consent of Joel Barnum of Descriptor Systems.

All trademarks belong to their respective companies.

You can contact Descriptor Systems at:

http://www.descriptor.com
info@descriptor.com
319-362-3906
P.O. Box 461
Marion, IA 52302

Full Stack Web Development for Liberty Mutual Labs

Lab 1: Introduction to JavaScript

Part 1: Working With JavaScript

Part 2: More Practice

Lab 2: JavaScript Data Types

Part 1: JavaScript Datatypes

Part 2: More Practice

Lab 3: JavaScript Operators and Flow Control

Part 1: JavaScript Operators

Optional Part 2: More Operators Practice

Part 3: JavaScript Flow Control Statements

Optional Part 4: More Flow Control Practice

Lab 4: JavaScript Objects, Arrays and Functions

Part 1: JavaScript Objects

Part 2: JavaScript Arrays

Part 3: JavaScript Functions

Lab 5: Introduction to React

Part 1: Getting Started with React

Part 2: Writing a React Component with Axios for AJAX

Lab 5 Bonus: React Components

Part 1: Generating a React Project

Part 2: Writing Stateful Root and Stateless DisplayFlier Components

Part 3: Writing a Stateful Form Component

Lab 6: Introduction to Spring

Part 1: A Simple Spring Application

Part 2: Java Configuration

Part 3: Using XML Configuration

Lab 7: Introduction to Spring Boot

Part 1: A Simple Spring Application

Part 2: A Spring Boot Web Application

Lab 8: Spring Annotations

Part 1: Getting Started

Part 2: Using Annotations for Dependency Injection

Lab 9: Spring Java Configuration

Part 1: Getting Started

Part 2: Using Java Configuration

Optional Challenge: Using the Groovy DSL

Lab 10: This Lab Intentionally Left Blank

Lab 11: This Lab Intentionally Left Blank

Lab 12: Spring and the Web

Part 1: Spring MVC with Java Configuration

Lab 13: Spring and JDBC

Part 1: Setting Up the Environment

Part 2: Setting Up the Web Project

Part 3: Spring JdbcTemplate DAO

Part 4: Web Front End for the DAO

Part 5: Completing the Application

Optional Lab 14: Spring Transactions

Part 1: Setup

Part 2: Writing the DAOs

Lab 15: Hello, World Web SOAP Service

Part 1: Creating a Web Service

Lab 16: Simple RESTful Service with JAX-RS

Part 1: Creating the Service

Optional Lab 17: RESTful CRUD Service using JAX-RS

Part 1: Creating the Service

Part 2: Implementing the R in CRUD

Lab 18: Spring MVC RESTful Service

Part 1: Creating the Service

Part 2: Implementing CRUD and Writing a React/axios Ajax Client

Lab 19: Additional Spring Boot Topics

Part 1: Working with Starter Dependencies

Part 2: Working with Autoconfiguration and Deployment

Part 3: Using an External Database

Part 4: Deploying as a WAR

Case Study Lab

Lab 1: Introduction to JavaScript

In this lab, you will write some simple JavaScript.

Objectives:

- To practice JavaScript

Part 1: Working With JavaScript

Steps:

_1. Ask your instructor for the following information:

Lab Installation directory: _____

Eclipse Installation directory: _____

Tomcat Installation directory: _____

If your computer has the standard lab setup, the directories will be something like:

```
Lab Installation directory:      c:\Users\username\webclass
Eclipse Installation directory:  c:\Users\username\java\eclipse
Tomcat Installation directory:   c:\Users\username\java\apache-tomcat-8.0.27
```

_2. Start Eclipse running:

a. Use Windows Explorer or My Computer to navigate to the **{Eclipse Installation Directory}**.

If your computer has the standard lab setup, the directory is:

```
c:\Users\username\java\eclipse
```

Note: Be sure to substitute your computer's actual username, e.g. *libtrain2* for the "username" above.

b. Start Eclipse running by double-clicking on *eclipse.exe*.

c. When prompted for the *Workspace*, browse to **{Lab Installation Directory}/workspace**, then press OK.

If your computer is running Windows Vista, Windows 7 or later and has the standard lab setup, the directory is:

```
c:\Users\username\webclass\workspace
```

Note: Be sure to substitute your computer's actual username, e.g. *libtrain2* for the "username" above.

d. Close the Welcome window by clicking on the 'X' in its titlebar.

We also recommend running Eclipse as a maximized window.

e. Configure Eclipse to use Chrome. From the menu, choose *Window - Web Browser - Default System Web Browser*.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

_3. Eclipse supports the notion of *perspectives*, which are sets of views (windows) that are relevant for a task, e.g. Java or Web development.

Ensure that you are in the *Java EE (default)* perspective by choosing from the Eclipse menu *Window - Perspective - Open Perspective - Other - Java EE (default)*.

_4. In these labs, we will use Tomcat as our application server. Configure the connection between Eclipse and Tomcat:

a. At the bottom of the Eclipse window, click the *Servers* tab.

b. In the Servers view, click the *No servers are available...* link to start the wizard.

c. In the Server Type list, expand the *Apache* category and click on *Tomcat v8.0* server. Press Next.

d. On the next page, click the Browse button and navigate to **{Tomcat Installation Directory}**. On Windows Vista or later, this directory will be something like (the last part, the sub-version number, may differ slightly):

```
c:\Users\username\java\apache-tomcat-8.0.27
```

Press Finish.

_5. Next, create a Dynamic Web project for this lab:

a. From the Eclipse menu, choose *File - New - Dynamic Web Project* to start the wizard.

b. On the first wizard page, for the *Project name*, enter **lab01**, then press Next.

c. On the next page, simply press Next.

d. On the final page, put a checkmark in the *Generate web.xml* box, then press Finish.

_6. Examine the *lab01* project in the *Project Explorer* window pane by clicking on the "+" symbol to expand it. Note the following:

- The *Java Resources: src* folder is where you would put your Java packages and code including servlets and JavaBeans.
- The *WebContent* folder is where you will put HTML, JSPs and so forth.
- The *WebContent/WEB-INF* folder is a special directory mandated by the JEE specification. It's where you put any configuration files and other files that you don't want accessible to the browser.

_7. Right-click on the WebContent folder choose *New - HTML File* to create a file named **hello.html**. On the wizard's second page, choose the *New HTML File (5)* template.

Eclipse opens the page into the editor.

_8. Within the *head* element, enter the following:

```
<script>
   alert("Hello, world");
</script>
```

_9. To test, in the Project Explorer, right-click on hello.html and choose *Run As - Run on Server*, restarting the server if prompted. The server should start and Eclipse will then start a Web browser with your page.

You should see the JavaScript "alert" dialog box with the "Hello, world" text. Click the OK button to dismiss it.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

_10. In the Project Explorer, right-click on hello.html and choose Copy, then right-click on WebContent and choose Paste, renaming the copy to **error.html**.

_11. Open error.html into the editor, and delete the closing quote (") after Hello, world.

Save the file, then run it on the server. What happens?

_12. Browsers typically do not display script errors, with the motivation being that they're not of much help to end users. But the Chrome browser ships with a set of Developer Tools, one of which is the JavaScript Console, in which you can see script errors.

To surface the Console, click the Chrome menu, which is at the top-right of the browser (its icon is three horizontal bars), then choose *More tools - Developer tools*.

The Developer tools will open at the bottom of the Web page. At the top of the tools pane, there's a menu - click the Console entry to display the JavaScript Console.

The Console will display a Syntax Error, complete with a line number (there also may be a 404 error for favicon.ico, which you can ignore).

Note: To make debugging scripts easier, we recommend that you enable line numbers in the Eclipse editor. In Eclipse, in the editor for error.html, right-click in the gray left margin (gutter) and choose *Show Line Numbers*.

In Eclipse, find and fix the error, save then refresh in Chrome. This is the basic development process for JavaScript.

Note: When refreshing pages with scripts, we recommend that you press Ctrl+Shift+R, which causes Chrome to do a "hard" reload to ensure a complete reload.

_13. Now let's use an external script:

a. In Eclipse, in the Project Explorer, right-click on hello.html and copy and paste it to **hello-external.html**.

b. Right-click on the WebContent folder and choose *New - File* to create **hello.js**.

c. Move the *alert* line (just the alert) from hello-external.html to hello.js.

d. Modify hello-external.html so it references the external script file.

e. Run hello-external.html on the server. It should work as before.

_14. Modify hello.js so it also writes the document's last-modified date to the page.

Part 2: More Practice

Steps:

_1. Create a new HTML file named **person.html**.

The page should have a button, that when clicked, uses the DOM to write your name, address and phone number to the page. Do not use document.write(). The page should use an external JavaScript file.

Test your work.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 2: JavaScript Data Types

In this lab, you will work with JavaScript datatypes.

Objectives:

- To use JavaScript data

Part 1: JavaScript Datatypes

Steps:

_1. Use *File - Close All* to close all open editing windows.

_2. In Eclipse, create a new Dynamic Web Project named **lab02**.

_3. In the WebContent folder, create a new HTML File named **datatypes.html**, and in it, write a script that:

- Defines a variable of each of these types: **number**, **string**, **boolean**, **Date**.
- Uses *document.write* to output each to the page.

Test your work.

_4. Create a new HTML File named **quote-of-the-day.html**. Code this file so it references an external JavaScript file that displays the following as a JavaScript alert (not in the page):

```
Today's quote is:

"On the Internet, nobody knows you're a dog."
```

Define only one variable and be sure to include the blank lines and quotes as shown.

_5. Create a new HTML page named **undefined.html** and an external JavaScript that defines a variable without initializing it and then displays its contents using document.write().

Part 2: More Practice

Steps:

_1. Create a HTML file named **variables.html** that references an external Javascript. In the script, do the following:

```
- declare a variable named z initialized
  to "abc"
- print z
- redeclare z and initialize to -876.554
- print z
- print a nonexistent variable q
```


Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Run and test, and use the JavaScript Console to see any error messages.

_2. Optional: Create a new HTML file named **equalityoperators.html** and an external JavaScript that demonstrates the difference between the == and === operators (we will cover them in more detail in a later chapter), perhaps using the examples in the Data Types chapter as inspiration.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 3: JavaScript Operators and Flow Control

In this lab, you will work with JavaScript operators and flow-control statements.

Objectives:

- To use JavaScript operators
- To use JavaScript flow-control statements

Part 1: JavaScript Operators

Steps:

_1. Use *File - Close All* to close all open editing windows.

_2. In Eclipse, create a new Dynamic Web Project named **lab03**.

_3. In the WebContent folder, create a new HTML File named **math.html** that uses an external JavaScript named **factorial.js** to calculate 10 factorial. Write the result to the page using the DOM (do not use document.write()). **Hint:** write the *script* tag after the *body* element.

_4. Update the last part to use another external JavaScript named **compound-interest.js** so that an element on the page also displays the future value of an investment given this formula:

```
P = C(1 + r/n) to the power of nt

P = Future value
C = Initial deposit
r = Interest rate as a percentage fraction (e.g. 0.075) is 7.5
n = number of times interest calculated per year
t = number of years
```

Note: You can use the *Math.pow(number, power)* function to raise a number to a power.

Use values of:

```
C = 10000
r = 0.075
t = 5
n = 4
```

Given those values, the result should be 14499.480257195399.

_5. Update the last part so it references another external JavaScript named **divide-by-zero.js** so it displays the result of dividing 100 by 0.

_6. Update the last part so it references another external JavaScript named **conversions.js** that implements the quiz on the last page of the Operators chapter. Your script should display the result of each question to the page.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Optional Part 2: More Operators Practice

Steps:

_1. Write a JavaScript program to solve the quadratic formula:

a. Google the "quadratic formula" - it solves equations of the form a(x**2) + bx + c = 0 and results in two answers (the formula has a + or - in it). Study the equation and think about how you could implement it.

b. Create a new HTML file named **quadratic.html** with an internal Javascript in the *head*. The script should contain a function named solvequadratic that accepts a, b and c:

```
function solvequadratic(a,b,c)
{
}
```

c. Complete the function so that it calculates both roots of the quadratic formula given the input values and displays the two roots in elements on the page (e.g. *div* elements).

In the body, create an HTML form with three text input widgets that let the user enter a, b and c as well as a submit button. When the user submits the form, invoke the function, passing whatever the user entered into the three input widgets.

d. Test with values 1, 3 and -4. The roots should be 1.0 and -4.0.

Part 3: JavaScript Flow Control Statements

Steps:

_1. Use *File - Close All* to close all open editing windows.

_2. In the WebContent folder, create a new HTML File named **who-are-you.html** with an internal JavaScript.

Complete the script so it:

a. Uses the prompt() function to ask the user to enter their name:

```
var name = prompt("Enter your name");
```

b. Uses an *if-else if- else* statement:

If the name is "Sam", write to the document "Hello, Sam!"

If the name is "Sue", write to the document "Goodbye, Sue!"

Any other name, write "Who are you?"

_3. Create a new HTML File named **measure-distribution.html** that uses an external JavaScript that generates 100,000 random numbers and counts how many are greater than 0.5, how many are less than 0.5 and how many are exactly 0.5. Display the results to a Web page using the DOM (do not use document.write()).

Hint: The Math.random() method returns a number between 0 and 1.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Optional Part 4: More Flow Control Practice

Steps:

_1. Write a JavaScript program that prompts the user for a string. Your script's job is analyze the characters in the string. Count the number of spaces, punctuation characters and "normal" characters. Display the counts in the Web page.

Hint: Strings have a charAt() method that returns the character at an index in a string. Indexes start at zero.

_2. Write an HTML form and Javascript that lets the user guess a number between 1 and 1000 with a maximum of 10 guesses. The form should contain a text input widget into which user can enter the guess and a submit button. On submission, tell the user whether guess is too high, too low or correct. The page should show the user how many attempts they have left.

To get a random number between 1 and 1000:

```
var theNumber = Math.floor(Math.random() * 1001);
```

Extra credit: Disable the submit button after 10 guesses.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 4: JavaScript Objects, Arrays and Functions

In this lab, you will work with JavaScript objects, arrays and functions.

Objectives:

- To use create objects and add/delete properties
- To work with arrays
- To work with various syntaxes to create and call functions

Part 1: JavaScript Objects

Steps:

_1. Use *File - Close All* to close all open editing windows.

_2. In Eclipse, create a new Dynamic Web Project named **lab04**.

_3. In the WebContent folder, create a new HTML file, create-objects-literal.html and a JavaScript that uses a literal object syntax to create an object named person with the following properties and values of your choice (address has nested properties):

```
firstname
lastname
age
address
    street
    city
    state
    zip
```

The script should build and then write a string containing the object's property values to the page.

_4. Copy create-objects-literal.html to a new file, **create-objects-new.html**. Rewrite the script to use "new" operator instead of literal syntax. Use a mixture of dot notation and [] notation to add the properties.

_5. Copy either of the first two pages to a new file, **enumerate.html**.

Add code so that it loops and enumerates all properties to display the property name and value to the page. Use a nested loop to display the nested "address" property.

Part 2: JavaScript Arrays

Steps:

_1. Use *File - Close All* to close all open editing windows.

_2. In the WebContent folder, create an HTML named **arrays.html** and a JavaScript that creates an array containing all of the months of the year. Then write a loop that prints all of the month strings to the document.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

_3. The document.getElementsByTagName() method returns an "array-like" object containing a list of all of the elements in the document with a given tag name. You can treat the array-like object just like an array, indexing as normal.

Create a new file, **paragraphs.html** that contains several HTML "p" elements with text in them. In a script at the end of the *body* element, use getElementsByTagName() to retrieve the array-like list of "p" elements, then iterate the array-like object, concatenating the text from each paragraph into a string. Then use alert() to display the string.

_4. Copy arrays.html to a new file, **array-methods.html**.

Use the array functions to do the following:

- Use slice to create a copy of the array, then reverse the order on the copy, then display copied array's elements to the document.
- Repeat the above, but also display the months sorted alphabetically.

Part 3: JavaScript Functions

Steps:

_1. Use *File - Close All* to close all open editing windows.

_2. Create an HTML file named **functions.html**. In it, using the **function declaration** syntax, write a function named **countCharacter**.

The function should accept two parameters - a string and a character and should return the count of the specified character in the string, case insensitive. Treat the second argument as optional - if not provided, search for the character 'a'.

Call the function with a string and a character of your choice, then display the results to the page.

_3. Using a **function definition expression**, define another function in a variable named **longestName()**. The function should accept an array of name strings and return the name with the most characters. If there's more than one with the longest name, return the first of such.

Call the function via the **longestName** variable and display the result to the Web page.

_4. Define an array of objects - each object represents an employee with a **employeeID**, **name** and **salary**.

Google "mdn array.sort" and read about the Array.sort() function. Note that it accepts an optional **comparison function** parameter. Read the docs, then use an anonymous function to compare employees so they are sorted in descending salary order. Display the sorted array to the Web page in an unordered list.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 5: Introduction to React

In this lab, you will learn the basics of the React JavaScript framework.

Objectives:

- To get started with the React framework

Part 1: Getting Started with React

Steps:

_1. Use *File - Close All* to close all open editing windows.

_2. In Eclipse, create a new Dynamic Web Project named **lab05part1**.

_3. In this and later labs, you will pretend that you work in the IT department of an airline and are writing an application to manage the airline's frequent-flier program. In this part, you will work with a *Segment* domain class that represents a single leg of air travel. A Segment has the following properties:

- **segmentNumber** - the unique ID for a given segment
- **segmentDate** - the date the segment was flown
- **flightNumber** - the airline's flight number for the segment
- **originatingCity** - three-letter airport code
- **miles** - mileage credited to traveler's frequent-flier account

_4. In the WebContent folder, create a new HTML file named **view-segment.html**. Complete the file:

a. In the *head*, write three *script* elements that reference React itself, React-DOM and babel-core. See an example in the course notes. Note that you can copy and paste these elements from **{Lab Installation Directory}/starters/lab05/starters.txt**.

b. Within the *body*, define an empty *div* element with a unique ID attribute that will act as the React target.

c. After the target div, write a script element with a type attribute of **type="text/babel"**. The Babel transpiler will transpile anything within this element into highly-compatible JavaScript that will run in a reasonably modern browser. In other words, Babel will convert any JSX syntax or EcmaScript v6 syntax into EcmaScript v5, which is highly supported by current browsers.

You will complete this script element in the next step.

_5. Complete the Babel-aware script element:

a. Define a JavaScript object that represents a Segment of air travel, using the description of the Segment domain class from an earlier step. Use values and types of your choice.

b. Define a React component that uses ES6 *arrow syntax*. The component should accept a reference to the Segment and display each of its properties in a separate paragraph.

Note: Be sure to name your component with an upper-case first character so ReactDOM.render() will not think you are trying to render a standard HTML element.

c. Call ReactDOM.render(), specifying your component's name and passing a reference to your Segment object. Render the virtual DOM into the target *div* you defined earlier.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

_6. Run view-segment.html on the Tomcat server, restarting Tomcat if prompted. Use the Chrome Developer Tools to help debug if you run into problems.

_7. Next, let's view the transpiled JavaScript source. In Chrome, open a new tab and navigate to **https://babeljs.io/repl**.

In Eclipse, select all of the lines within the Babel *script* element and copy them into the clipboard. Then back in Chrome, paste the clipboard into the left-side text area.

Examine the right-side text area, which shows the transpiled JSX and arrow functions. Especially note how Babel calls React.createElement for each JSX element.

Part 2: Writing a React Component with Axios for AJAX

In this part, we will provide you with a prewritten RESTful service that provides frequent-flier information in JSON. You will create a React component that displays the JSON data.

Steps:

_1. Your first job is to generate an Eclipse project for the prewritten RESTful service. We will use the Gradle build tool to satisfy dependencies and generate the project. Follow this procedure:

a. In the Windows Start menu, at the bottom, enter **cmd** and press Enter to open a command prompt window.

b. Change to the lab's directory. If your computer has the standard lab setup:

```
cd \Users\username\webclass\workspace\lab05service
```

c. We have provided you with a Gradle build script that will download the dependencies for your project and generate an Eclipse project. Open it and see if you can make any sense of it (we will learn more about Gradle later).

```
write build.gradle
```

Note that the prewritten application uses the Spring Framework to provide the RESTful data.

d. Back in the command prompt, generate an Eclipse project and download the dependencies:

```
gradle eclipse
```

_2. Your next job is to import the project into Eclipse. Follow this procedure:

a. From the Eclipse menu, choose *File - Import* to start the wizard.

b. On the first wizard page, expand the *General* category and highlight *Existing projects into workspace*, then press Next.

c. On the next wizard page, click the Browse button next to the *Select root directory* button - Eclipse will show the *workspace* folder. Just click *Open* to select that folder.

d. In the project list, ensure that ONLY *lab05service* is selected. Then look below the project list and ensure that the *Copy projects into workspace* checkbox is NOT selected. Press Finish to import the project.

_3. Examine the newly imported project:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

a. The *src/main/java* folder contains the Java artifacts that comprise the RESTful application. You should especially study the two classes in the *com.example.demo.ffprogram.domain* package - they define the domain model. Flier.java represents a member of the frequent-flier program and ContactInfo.java contains the flier's address and phone numbers. The prewritten RESTful application exposes data from these two classes in JSON format.

_4. Before writing your React code, let's do a quick test to ensure that the RESTful service is working:

a. Back in your command prompt, boot the application into an embedded Tomcat server:

```
gradlew bootRun
```

Note: To determine that the server has started, look in the command-prompt for the text "Started DemoApplication...". The progress percentage will not reach 100%.

b. Open a Chrome tab and browse to this URL:

```
http://localhost:8888/fliers
```

You should see some JSON. Study this output carefully - your React application will display this data in an HTML table.

_5. Back in Eclipse, create a new Dynamic Web project named **lab05part2**.

_6. We recommend that you create a file in the lab05part2 project's root directory named **fliers.json**, then copy and paste the JSON from the browser into the file. Then, from the Eclipse menu, choose *Source - Format*. Note that the JSON is organized as an array of objects, each containing Flier data. Furthermore, each Flier contains a sub-object with ContactInfo. Look back at the Java domain classes and ensure you understand this domain model.

_7. In the lab05part2 project's *WebContent* folder, create an HTML file named **view-fliers.html**. Start working on this React page:

a. In the *head*, write script elements that reference:

- React
- React-DOM
- Babel core
- axios

b. In the *body*, define a *div* element that will act as the target of React's virtual DOM updates.

c. Still in the *body*, write a *script* element with type of **text/babel**. You will put your React code in this block.

_8. Complete the Babel script:

a. Within the Babel script block, establish the initial content of the target *div* by calling ReactDOM.render((), rendering a child *div* with the text "Waiting for response from server".

b. Define a React component named **FlierList** using an ES6 arrow function. This component will be passed a *props.fliers* parameter, which is the array of Fliers in JSON format. This component's job is to display the JSON array as an HTML table. You should look back at the JSON content to re-acquaint yourself with it.

To iterate the table rows (*tr* elements), use the Array.map() function, writing an anonymous function that accepts a reference to a Flier. **Note:** React requires that each table row has a unique key - you can use the Flier ID from the JSON for the key.

We recommend that at minimum, each table row display:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

- Flier name
- Membership level (e.g. Gold)
- Flier's email address from the nested *contactInfo*

c. Use *axios.get()* to send an AJAX request to **http://localhost:8888/fliers** to retrieve the JSON. The get() method returns a *promise* on which you can register two functions. The first registration function is named **then()** - it represents a successful response. The second registration function is named **catch()** - it represents errors.

Call both then() and catch(), registering anonymous callback functions for "success" and "error".

In your "success" anonymous function, retrieve the JSON Flier array from the response's *data* property, then call ReactDOM.render() to re-render the FlierList component into the target *div*. Pass the JSON array to the component as an attribute named **fliers** - recall that the component expects this array in its *props.fliers* parameter.

In your "error" anonymous function, simply log the error to the JavaScript console.

_9. Run view-fliers.html on the server. You should see the Flier list in tabular form. If you have issues, use the Chrome Developer Tools to help you debug.

_10. Back in the command prompt, press Ctrl+C to stop the service.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 5 Bonus: React Components

In this lab, you will work with React components, their state and lifecycle. You will also use the *create-react-app* script to bootstrap the application.

Objectives:

- To learn about React components' props, state and lifecycle
- To use a more real-world React development process

Part 1: Generating a React Project

Steps:

_1. Use *File - Close All* to close all open editing windows.

_2. You will use React's *create-react-app* script to generate a starter application for your React application:

a. Open a command prompt window and change to the *workspace* directory:

```
cd \Users\username\webclass\workspace
```

b. Use the *create-react-app* script to generate a starter project named **lab05Z**:

```
create-react-app lab05Z
```

Note that this may take several minutes for the script to download all of the packages for React development into a local *Node.js package manager* (npm) repository.

The script will create a new sub-folder named *lab05Z* and create the starter application there.

c. As in the last lab, we will use the *axios* library for Ajax. The *create-react-app* script doesn't install it, but it's easy to load packages into the npm repository:

```
cd lab05Z
npm install axios
```

You can ignore warnings about skipping optional dependencies.

d. Use Windows Explorer or My Computer to browse the *lab05Z* folder. Especially examine the *src* folder and note the generated files (you don't need to open them yet):

- **App.css** contains some simple CSS rules for the generated React component
- **App.js** contains a starter React component
- **App.test.js** contains a simple unit test for the component
- **index.css** contains some simple CSS rules for the entire application
- **index.js** renders the starter component into the page DOM

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

- **logo.svg** is a picture used by the generated app
- **registerServiceWorker.js** bootstraps the application into a web server and performs various other functions

_3. Back in the command prompt, you can start the generated application to see the provided function:

```
npm start
```

This will bootstrap the application and start a development web server, load the starter application and then launch your computer's default browser.

_4. The web server is "live", that is, if you modify and save a file, it will update the page automatically. Let's see how it works:

a. Open a new command prompt window and change to the *lab05Z\src* folder.

b. Use Windows WordPad to make a quick change:

```
write App.js
```

Find the *h1* element that says "Welcome to React" and change the text in some way, then save the file.

Resurface the browser to see the change. This is the basic development process when using create-react-app. Of course, using Windows WordPad is not ideal, and we will address that in the next part!

c. Close WordPad.

Part 2: Writing Stateful Root and Stateless DisplayFlier Components

Eventually, your application will comprise three React components:

- A stateful root component named *App* that stores the app's state and passes it down the tree. The app's state will be the data for the Flier being currently displayed
- A stateless component that displays a Flier. It will accept the current Flier as *props*
- A stateful component with a form that lets the user find a Flier by ID. When the user submits the form, this component will invoke a method in the root component to update the current Flier

In this part, you will code the stateless DisplayFlier component and start working on the root App component.

Steps:

_1. This lab will use the same frequent-flier service as lab05. Review lab05 and use *gradlew bootRun* to start the service if it's not already running.

_2. Verify that you can access a specified Flier with a URL that ends in the Flier's ID. Open a web browser and enter:

```
http://localhost:8888/fliers/2
```

You should see JSON for "Joe Jones".

Note the pattern for URLs: the base URL for the REST service is *http://localhost:8888/fliers*. To specify a given Flier, put the flierID after a slash.

_3. You will use the Atom editor in this lab to create the web front-end to your application:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

a. There should be a shortcut to Atom on your desktop - click it to launch Atom.

You can close any untitled, Welcome and Telemetry Consent tabs.

b. From the Atom menu, choose *File - Open Folder*, then browse to **{Lab Installation Directory}/workspace** and select the **lab05Z** folder.

Atom opens the folder as a project and displays a directory tree.

_4. Start by creating a stateless React component that displays a single Flier that's passed to the component in *props*:

a. From the Atom menu, choose *File - New File* to create a new untitled file, then *File - Save* to save it in the *src* folder as **DisplayFlier.js**.

b. The create-react-app script generates files organized as ES6 modules. This new file will define an export the DisplayFlier component.

To get started, at the top of the file, import the React module itself:

```
import React from 'react';
```

c. Using an ES6 *arrow function* and *object destructuring* syntax to reference *props*, define the DisplayFlier component's skeleton:

```
const DisplayFlier = ({currentFlier}) =>
{


};
```

d. Within the arrow function's body, *return* JSX with:

- A root HTML5 *section* element enclosing the component's markup
- Within the *section*, write an *h2* element with text **Current Flier** and *div* elements that display the current Flier's *flierID*, *flierName*, *level* and *contactInfo.emailAddress* properties

You can use the DisplayStudent component in the Complete Example at the end of the React Components chapter for inspiration.

e. At the end of the file, after the arrow function, export the component so it's available to other components:

```
export default DisplayFlier;
```

_5. Though the app's not complete yet, open App.js in Atom and modify it so it uses the DisplayFlier component to display a hard-coded Flier:

a. At the top of App.js, you can delete the *imports* for logo.svg and App.css since we will not use them.

b. Write a new *import* to bring in the DisplayFlier component:

```
import DisplayFlier from './DisplayFlier.js';
```

c. Above the *render()* method, but within the App component, write a constructor that initializes state for a hard-coded Flier:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
constructor(props)
{
  super(props);

  this.state =
  {
      currentFlier:
      {
          flierID: -1,
          flierName: "No name",
          contactInfo:
          {
              emailAddress: "No email"
          }
      }
  };
}
```

d. Find the *render()* method and delete all of the JSX markup within the *return()*.

e. Within the *return()*, emit JSX markup:

```
<section>
  <h1>Flier Information</h1>
  <div>
     <DisplayFlier currentFlier={this.state.currentFlier}/>
  </div>
</section>
```

Note the architecture here - App is the root component and stores the application's state. DisplayFlier is a stateless component, and App passes the current Flier in *props*.

f. Save the file and then resurface the browser - you should see the hard-coded Flier info from your DisplayFlier component.

_6. Next, update App.js so it fetches the Flier list from the REST server and uses the first returned Flier as the "current" Flier:

A good place to get initial data for the root component is in the component's *componentDidMount()* lifecycle method.

a. In App.js, near the top, add another import to bring in the *axios* Ajax library:

```
import axios from 'axios';
```

b. In App.js, write an empty *componentDidMount()* method that accepts no arguments.

c. Within the method, use *axios.get()* to retrieve the JSON Flier list from **http://localhost:8888/fliers**. In the promise's "success" function, store *response.data* into a variable named **flierList**, then call setState():

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
const flierList = response.data;

this.setState(
{
    currentFlier: flierList[0]
});
```

Note that we call setState() rather than directly modifying the currentFlier variable directly - this ensures that React can re-render any descendent components, in this case, the DisplayFlier component.

You should also write an "error" callback function on the promise and log a descriptive message to console.log().

d. Save App.js - the browser should update and display information about the first Flier, "Sue Smith".

Part 3: Writing a Stateful Form Component

In this part, you will complete the application by writing a stateful form component that lets the user find a Flier by ID. If successful, the root component will call setState(), thus re-rendering the DisplayFlier component.

Steps:

_1. Start by creating a new file named **FindFlierForm.js**. In it, be sure to import React as you did in DisplayFlier.js.

_2. In FindFlier.js, write the skeleton a component named FindFlierForm. This will be a stateful component, so it should be a class that extends React.Component.

_3. Within the class, write the skeletons of four methods:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
constructor(props)
{
    super(props);
}

handleInputChange(event)
{

}

handleSubmit(event)
{

}

render()
{
  return(
    <section>
      <h2>Find Flier</h2>
      <form>
      </form>
    </section>
  );
}
```

_4. At the bottom, after the end of the class, export the component:

```
export default FindFlierForm;
```

_5. To ensure you are on the right track, update the root component, App.js so that the new component will be verified in the live development server:

a. In App.js, near the top, import FindFlierForm in the same fashion as you did for DisplayFlier. Save the files.

b. In App.js, in the render() method, after the close *div* for the DisplayFlier component, write another *div*, and within it, reference the FindFlierForm component:

```
<FindFlierForm />
```

c. Look in the browser - the new component is not complete, but there should be no errors. If there are, fix them before continuing.

_6. Next, flesh out the markup for the form in FindFlierForm.js. In the render() method, within the *form* element:

a. Define a label and an *input* element so the user can enter the Flier ID. The *input* element should have the following attributes:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
Attribute    Value
---------    ------

name         flierID
type         number
onChange     {this.handleInputChange}
value        {this.state.flierID}
```

Note how the *onChange* attribute references one of your as-of-yet empty methods and the *value* attribute references a state variable that you will initialize in a later step. Also be aware the an input element's *value* is read-only, so we are not violating the rule to only modify state by calling setState().

Note: If you save the file at this point, you'll get an error complaining about "Cannot read property 'flierID' of null". That's because you haven't defined the component's state yet. You will address this shortly.

b. Write another *input* element with the following attributes:

```
Attribute    Value
---------    ------

type         submit
onClick      {this.handleSubmit}
value        Find
```

Note how the *onClick* attribute references another empty method.

c. Still within the form, after the "submit" input element, define a placeholder for an error messages resulting when the user submits the form:

```
<span className="errorMsg">
    {this.props.message}
</span>
```

_7. Next, update the FindFlierForm's constructor:

a. As you did in the App.js constructor, establish a state variable named **flierID** and initialize it to -1.

b. Ensure that within the handleInputChange and handleSubmit methods, the *this* reference is for the component, not the function itself:

```
this.handleInputChange = this.handleInputChange.bind(this);
this.handleSubmit = this.handleSubmit.bind(this);
```

Save the file - you should have no errors at this point, though you still have some work to do. You should see the form on the page.

_8. Next, complete the handleInputChange method. The good news is that there's a standardized way to code this method that should work with any form:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
const target = event.target;
const value = target.value;
const name = target.name;

this.setState(
{
  [name]: value
});
```

Note: This code assumes that when you wrote the JSX markup for the *input* elements where the user enters data, that the *name* attribute EXACTLY matches the name of a state variable, in this case, *flierID*.

_9. Next, complete the handleSubmit method:

```
// invoke the callback function from this component's ancestor
if(this.props.onFlierSelected)
{
    this.props.onFlierSelected(this.state.flierID);
}
else
{
    console.log("Error: parent must provide onFlierSelected callback");
    console.log("Selected flier ID: " + this.state.flierID);
}

event.preventDefault();
```

Note how the parent component, in this case App.js, must provide a reference to a callback function that will accept the flier ID that the user enters. Please think about how this architecture works: the state of "current" Flier is maintained by the root component, so when the user interacts with the FindFlierForm descendent component, it must notify the root so it can modify the state.

After saving, you can test your work so far. Press the Find button in the form, then look in the browser's console - you should see the "parent must provide" error message.

_10. Next, start updating App.js to fully support the FindFlierForm component:

a. Write an empty method that will act as a callback when the user submits the form:

```
flierSelected(flierID)
{

}
```

b. In the App.js constructor, define another state variable named **findFlierMessage** and initialize it to an empty string.

c. Still in the constructor, similarly as you did in the FindFlierForm, ensure that the component's *this* is bound within the flierSelected method.

_11. Next, complete the flierSelected() method:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

a. Use *axios.get()* to retrieve the selected Flier's data. The URL should be of the form http://localhost:8888/fliers/{flierID}, replacing the {flierID} with the actual numeric flier ID that's passed to this method as a parameter.

b. In the "success" callback, call setState() to set the *currentFlier* state property to the data returned by the Ajax request and the *findFlierMessage* state property to an empty string.

c. In the "error" callback, call setState() to set the *findFlierMessage* state property to the error returned from the Ajax request:

```
this.setState(
{
    findFlierMessage: error.message
});
```

d. In the App.js render() method, modify the FindFlierForm element so it has the following attributes:

```
Attribute        Value
---------        -----

message          {this.state.findFlierMessage}
onFlierSelected  {this.flierSelected}
```

_12. Save all your files and then test your app. You should be able to successfully retrieve the Flier with ID of 2, but if you try finding Flier #22, your application should display an error message.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 6: Introduction to Spring

In this lab, you will write a simple Spring application.

Objectives:

- To write a Simple Spring application

Part 1: A Simple Spring Application

In this part, you will create and run a simple Spring application.

Some of the lab projects for this course will focus on an application that implements a frequent-flier program for a fictitious airline, Oak Tree Airlines. The application will manage the frequent-flier program's members, their mileage, bonuses and awards.

Steps:

_1. Your first job is to generate an Eclipse project for this lab. We will use the Gradle build tool to satisfy dependencies and generate the project. Follow this procedure:

   a. In the Windows Start menu, at the bottom, enter **cmd** and press Enter to open a command prompt window.

   b. Change to the lab's directory. If your computer has the standard lab setup:

      ```
      cd \Users\username\webclass\workspace\lab06
      ```

   c. We have provided you with a Gradle build script that will download the dependencies for your project and generate an Eclipse project. Open it and see if you can make any sense of it:

      ```
      write build.gradle
      ```

      Note that it expresses a dependency on Spring version 4.2.6 and specifies the *eclipse* plugin that knows how to generate Eclipse projects. Close WordPad when you are finished examining the file (you don't need to modify it).

   d. Generate an Eclipse project and download the dependencies:

      ```
      gradle eclipse
      ```

      You will note that if you've previously run Gradle, it doesn't need to download as much - that's because Gradle caches downloaded libraries.

_2. Your next job is to import the project into Eclipse. Follow this procedure:

   a. From the Eclipse menu, choose *Window - Perspective - Open Perspective - Java* to switch to the Java perspective - this lab will be a standalone Java application, not a web application.

   b. From the Eclipse menu, choose *File - Import* to start the wizard.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

c. On the first wizard page, expand the *General* category and highlight *Existing projects into workspace*, then press Next.

d. On the next wizard page, click the Browse button next to the *Select root directory* button - Eclipse will show the *workspace* folder. Just click *Open* to select that folder.

e. In the project list, ensure that ONLY *lab06* is selected. Then look below the project list and ensure that the *Copy projects into workspace* checkbox is NOT selected. Press Finish to import the project.

f. In the Project Explorer, expand the project and note that it contains the build.gradle script and that the Referenced Libraries folder contains several JARs.

g. Right-click on the project and choose *New - Source folder* and create a folder named **src/main/java**. This is where you will put your Java code.

_3. Right-click on the project's **src/main/java** folder and choose *New - File* to create a new file named **spring.xml**. Press the *Source* tab at the bottom of the editor window and then add the following contents. You can copy and paste from the **{Lab Installation Directory}/starters/spring/lab06/spring.txt** file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:aop="http://www.springframework.org/schema/aop"
 xmlns:context="http://www.springframework.org/schema/context"
 xmlns:jee="http://www.springframework.org/schema/jee"
 xmlns:jms="http://www.springframework.org/schema/jms"
 xmlns:lang="http://www.springframework.org/schema/lang"
 xmlns:tx="http://www.springframework.org/schema/tx"
 xmlns:util="http://www.springframework.org/schema/util"
 xmlns:mvc="http://www.springframework.org/schema/mvc"
 xsi:schemaLocation="http://www.springframework.org/schema/aop
     http://www.springframework.org/schema/aop/spring-aop-4.2.xsd
     http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-4.2.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-4.2.xsd
        http://www.springframework.org/schema/jee
        http://www.springframework.org/schema/jee/spring-jee-4.2.xsd
        http://www.springframework.org/schema/lang
        http://www.springframework.org/schema/lang/spring-lang-4.2.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-4.2.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc-4.2.xsd
        http://www.springframework.org/schema/util
        http://www.springframework.org/schema/util/spring-util-4.2.xsd">




</beans>
```

Note: If necessary, press the *Source* tab at the bottom of the editor window so you can paste the "raw" XML.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

This defines the root element, *beans* of the Spring configuration file and all of the namespaces we will need.

_4. Start your Java work by creating a class that represents a segment of air travel:

a. In the Package Explorer, right-click on the *src/main/java* folder and choose *New - Class* to start the wizard.

b. On the first wizard page, for the *Package*, enter **com.oaktreeair.ffprogram**.

For the *Name*, enter **Segment**, then Press Finish.

c. Immediately above the "public class Segment" line, annotate the class so that it's a Spring component:

```
@Component("seg01")
```

Use the *Source - Organize Imports* menu to import *org.springframework.stereotype.Component*.

d. In the new class, define the following fields:

```
private Long segmentNumber;
private Date segmentDate;
private int flightNumber;
private String originatingCity;
private int miles;
```

Use the *Source - Organize Imports* menu to import *java.util.Date*.

e. Use the *Source - Generate Getters and Setters* menu to create get/set methods for all of the fields.

f. Use the *Source - Generate toString* menu to generate a toString() method that includes all of the fields.

Save to compile.

_5. Repeat the above step to create a class named **ContactInfo** in the *com.oaktreeair.ffprogram* package.

Complete the *ContactInfo* class:

a. Annotate the class so that it's a Spring component named "contact01".

b. Define fields:

```
private String emailAddress;
private String homePhone;
private String mobilePhone;
private String smsNumber;
```

c. Generate get/set methods for the fields. Also generate toString().

_6. Repeat the above step to create a class named **Flier** in the *com.oaktreeair.ffprogram* package:

a. Annotate the class so that it's a Spring component named "flier01".

b. Define fields:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
private String flierName;
private Long flierID;
private ContactInfo contactInfo;
public enum Level
{
    Member, Gold, Platinum
};

private Level level;
```

Generate get/set methods for the fields.

c. Write another field:

```
 private ArrayList<Segment> segments = new ArrayList<Segment>();
```

Import the *java.util.ArrayList* type, then generate ONLY a "get" method for the field.

d. Write another method that adds a Segment to the collection:

```
public void addSegment(Segment seg)
{
    segments.add(seg);
}
```

_7. Next, use annotations to initialize the Flier's fields:

a. Edit Flier.java and find the flierName field. Annotate the field to provide a value:

```
@Value("Sam Smith")
```

Use *Source - Organize Imports* to import the Value type.

b. Repeat the above to initialize the flierID field to "1234".

_8. Next, edit ContactInfo.java and repeat the above step to initialize the ContactInfo's fields to values of your choice.

_9. Next, use an annotation to configure a simple injection:

a. Edit Flier.java and find the ContactInfo field.

b. Annotate the field to inject the "contact01" bean you configured eariler:

```
@Resource(name="contact01")
```

Use the *Source - Organize Imports* to import the *javax.annotation.Resource* type.

_10. Create an interface for a "service" that calculates a bonus for a segment of air travel for a flier:

a. Right-click on the com.oaktreeair.ffprogram package and choose *New - Interface* to start the wizard.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

b. For the *Name*, enter **BonusCalc**, then press Finish.

c. Add the following methods to the new interface:

```
public int calcBonus(Flier flier, Segment seg);
public int calcBonus(Flier flier, Collection<Segment> segments);
```

Use *Source - Organize Imports* to import *java.util.Collection*, then save to compile.

_11. Create a class that implements the BonusCalc interface:

a. Right-click on the com.oaktreeair.ffprogram package and choose *New - Class* to start the wizard.

b. For the *Name*, enter **BonusCalcImpl**.

For the *Interfaces*, press the Add button, then start typing **Bonus** in the *Choose interfaces* entry box. Watch as Eclipse narrows down the type, then select *BonusCalc - com.oaktreeair.ffprogram* and press OK. Press Finish to complete the class creation.

Note that Eclipse created method stubs for the interface methods.

c. Annotate the class so that it's a Spring component named "calcBonus".

d. The rules for a bonus is that a "Member" flier receives a 10% bonus, a "Gold" flier receives 25% bonus and a "Platinum" flier receives 50%. The bonus is based on the miles in the segment.

Implement the first calcBonus method that accepts a single Segment, then if you have time, also implement the method that accepts a collection.

Your code for the first calcBonus method might look like:

```
@Override
public int calcBonus(Flier flier, Segment seg)
{
    double bonus = 0;
    double miles = seg.getMiles();

    switch(flier.getLevel())
    {
        case Member:
            bonus = miles * 0.10;
            break;
        case Gold:
            bonus = miles * 0.25;
            break;
        case Platinum:
            bonus = miles * 0.50;
            break;
    }

    return (int)bonus;
}
```


Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

_12. Edit the spring.xml file, and immediately above the *beans* end tag at the bottom of the file, write configuration elements so that Spring will scan to find the annotations

```
<context:component-scan base-package="com.oaktreeair.ffprogram"/>
```

_13. As before, but in the **com.oaktreeair.test** package, create a new class named **FrequentFlierProgram**. This class doesn't implement any interfaces, but should have a *main* method (click the appropriate checkbox in the New-Class wizard).

Complete the new class's *main* method:

a. Using the example in the notes as a guide, create a Spring *FileSystemXmlApplicationContext* that references the Spring configuration file with path **src/main/java/spring.xml**:

```
AbstractApplicationContext ctx =
        new FileSystemXmlApplicationContext("src/main/java/spring.xml");
```

Use the *Source - Organize Imports* menu to import the appropriate types.

b. Using the example in the notes as a guide, retrieve a bean named **flier01** from the container, storing the reference in a variable typed to the Flier interface:

```
Flier flier01 = . . .;
```

Import types as necessary.

c. Display the Flier's name and ID to the console using System.out.println:

```
System.out.println(flier01.getFlierName());
System.out.println(flier01.getFlierID());
```

d. Retrieve the Flier's ContactInfo and store the reference in a variable:

```
ContactInfo inf = flier01.getContactInfo();
```

Display the email address from the contact info to the console.

e. Try running the program at this point: In the Package Explorer, right-click on FrequentFlierProgram.java and choose *Run As - Java Application*.

You should get a couple of log4j warnings (we will fix that later) followed by the Flier and ContactInfo properties that you displayed. Does that make sense? Note how the @Value and @Resource annotations configured initial values and dependency injection.

_14. Edit Segment.java and write @Value annotations to initialize the fields to values of your choice. You can skip the segmentDate for now, but be sure to initialize the miles. We suggest using a value of "1000" to make it easier to determine if the bonus calculation is correct.

_15. Edit FrequentFlierProgram.java and test your bonus calculation:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

a. Set the flier01's Level to Flier.Level.Gold:

```
flier01.setLevel(Flier.Level.Gold);
```

b. Retrieve a "seg01" bean reference from the container.

c. Add the segment to the flier:

```
flier01.addSegment(seg01);
```

d. Retrieve a "calcBonus" bean reference from the container.

e. Call the calcBonus method, passing the flier01 and seg01 references, print the returned bonus value to the console:

```
int bonus = bc.calcBonus(flier01, seg01);
System.out.println("Bonus: " + bonus);
```

Run the FrequentFlierProgram as a Java application. If you set the seg01's miles to 1000, the bonus should be 250.

_16. If you mess something up, Spring throws exceptions. Reading the resulting "stack trace" to diagnose problems is an important skill. In this step, you will intentionally break the program to see how Spring responds.

a. Modify FrequentFlierProgram.java so it attempts to retrieve a bean named **flier123** from the container, then run. Examine the stack trace in the Eclipse console and ensure you understand it.

Comment-out the line of code above.

b. Modify Flier.java and change the @Value annotation for the *flierID* field to **abc**, then run the FrequentFlierProgram. Examine the stack trace in the Eclipse console and ensure you understand it.

Change the @Value back to a legal integer.

c. Modify Flier.java and change the @Resource annotation for the *contactInfo* field to **contact123**, then run the FrequentFlierProgram. Examine the stack trace in the Eclipse console and ensure you understand it.

Change the @Resource back to **contact01**.

_17. Now let's get rid of those pesky Log4J warnings:

a. Right-click on the *src/main/java* folder and create a new file named **log4j.properties**.

b. Enter the following into the new file:

```
log4j.rootLogger=WARN, stdout

log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d %p [%c] - %m%n
```

c. Run the FrequentFlierProgram class again -- the output should look a little cleaner.

_18. Now let's experiment with more verbose logging. Edit the *log4j.properties* file and change the first line so it looks like:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
log4j.rootLogger=FINE, stdout
```

Save the file, then run the main program again and examine the console output. Do you think the extra information could help you if things weren't working correctly?

Try changing the logging level to WARN and running the program again.

You can change the log4j level back to INFO if you wish.

Part 2: Java Configuration

In this part, you will convert the program so it use Java configuration instead of annotations.

Steps:

_1. Right-click on the lab06 project and choose *Copy*, then press Ctrl+V. Name the new project **lab06JavaConfig**.

_2. Edit Segment.java, Flier.java, BonusCalcImpl.java and ContactInfo.java and remove all of the annotations. You can use *Source - Organize Imports* to remove unneeded imports.

_3. Right-click on *com.oaktreeair.test* and create a new class named **FrequentFlierConfig**. Annotate the class with *@Configuration*.

_4. In FrequentFlierConfig.java, configure a Segment bean named *seg01*:

```
@Bean(name="seg01")
public Segment getSeg01()
{
    Segment seg = new Segment();
    seg.setFlightNumber(2238);
    seg.setMiles(1000);
    seg.setOriginatingCity("ORD");
    seg.setSegmentNumber(1234L);

    return seg;
}
```

_5. In FrequentFlierConfig.java, using the same technique as for Segment, configure a ContactInfo bean named **contact01**, by writing a bean-configuration method named **getContact01**. Call the bean's "set" methods to initialize the properties to values of your choice.

_6. In FrequentFlierConfig.java, configure a BonusCalcImpl bean named **calcBonus**:

```
@Bean(name="calcBonus")
public BonusCalc getCalcBonus()
{
    return new BonusCalcImpl();
}
```

_7. In FrequentFlierConfig.java, configure a Flier bean named **flier01**. Call the bean's "set" methods to initialize its properties to values of your choice. To establish the link from the Flier to its ContactInfo, you can use code like:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
f.setContactInfo(getContact01());
```

_8. Modify the FrequentFlierProgram class so it creates the Spring container using AnnotationConfigApplicationContext. Replace the existing container-creation code with:

```
AnnotationConfigApplicationContext ctx =
    new AnnotationConfigApplicationContext(
            FrequentFlierConfig.class);
```

_9. Run FrequentFlierProgram - it should work as before.

Part 3: Using XML Configuration

In this part, you will convert the program so it uses no annotations or Java configuration, but instead configures everything in XML.

Steps:

_1. Right-click on the lab06 project and choose *Copy*, then press Ctrl+V. Name the new project **lab06XML**.

_2. Edit Segment.java, Flier.java, BonusCalcImpl.java and ContactInfo.java and remove all of the annotations. You can use *Source - Organize Imports* to remove unneeded imports.

_3. Edit spring.xml and convert it so the beans, their properties and dependency injection is defined in XML:

a. Delete the *context:component-scan* element - you only need it if you're using annotations.

b. Define a Segment bean:

```
<bean id="seg01" class="com.oaktreeair.ffprogram.Segment">
  <property name="segmentNumber" value="1234" />
  <property name="flightNumber" value="2238" />
  <property name="originatingCity" value="ORD" />
  <property name="miles" value="1000" />
</bean>
```

Note how the property names match the "set" methods in Segment.java. When you use the *property* element in XML, Spring looks for the corresponding "set" method (it doesn't use the fields, only the "set" method).

c. Repeat the above step to define a "contact01" bean of the ContactInfo class, using property values of your choice.

d. Repeat the above step to define a "flier01" bean of the Flier class, using property values of your choice. For the *contactInfo* property, use the **ref** attribute to reference the "contact01" bean.

e. Repeat the above step to define a "calcBonus" bean of the BonusCalcImpl class. This bean needs no properties.

_4. Run FrequentFlierProgram as a Java application. It should work the same as in Parts 1 and 2.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 7: Introduction to Spring Boot

In this lab, you will start working with Spring Boot.

Objectives:

- To get started with Spring Boot

Part 1: A Simple Spring Application

In this part, you will create and run a simple Spring Boot application. The application will be a command-line application.

Steps:

_1. Your first job is to create the starter Spring Boot application. You will use the Spring Boot **Command Line Interface (CLI)** to generate the starter. Follow these instructions:

   a. The person who set up your computer should have already installed the Spring Boot CLI. To test it, open a command-prompt (Windows) or Terminal (OS X) and enter:

      ```
      spring --version
      ```

      You should see the Spring Boot CLI version number. If you don't, either ask your instructor or the person who set up your computer for this class.

   b. In the command-prompt, *cd* to the **{Lab Installation Directory}**. If your computer has the standard lab setup on Windows:

      ```
      cd \Users\username\webclass\workspace\lab07
      ```

   c. Use the Spring Boot CLI and Initialzr to generate a starter project:

      ```
      spring init -b 1.3.1.RELEASE --build gradle -x
      ```

      The *-x* options tells the CLI to expand the project from a ZIP file.

      The CLI will contact start.spring.io and then generate a project and unzip it. Use Windows Explorer or Finder to examine the project structure, comparing to the picture in the Intro to Spring Boot chapter in the course notes. It should be similar, but with different names.

_2. Use your Windows Wordpad to open **build.gradle**:

   ```
   write build.gradle
   ```

   Examine the file and ensure you understand it.

_3. In the command prompt or Terminal, generate an Eclipse project:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
gradlew eclipse
```

_4. Start Eclipse if it's not already running.

_5. Import the project:

a. From the Eclipse menu, choose *File - Import* to start the wizard.

b. On the first wizard page, expand the *General* category and highlight *Existing projects into workspace*, then press Next.

c. On the next wizard page, click the Browse button next to the *Select root directory* button - Eclipse will show the *workspace* folder. Just click *Open* to select that folder

d. Note that in the Project list, *lab07* is selected. Press Finish to complete the import.

e. Examine the project structure and files, especially *src/main/java/com/example/demo/DemoApplication.java*.

_6. Next, create the (small) business model for this application:

a. In the *src/main/java* folder, create a new package named **com.example.demo.shipping**, and in it, an interface named **Shipping** with the following methods:

```
double calculateDomestic(String zip);
double calculateInternational(String country);
```

b. In the same package, create a class named **ShippingImpl** that implements the interface. Simply return hard-coded values of your choice from the methods.

c. Annotate the class with **@Component** with **shipping** as the name.

_7. Create the "controller" that will act as the program's starting point:

a. In a new **com.example.demo.service** package, create a class named **ShippingController** that implements the *org.springframework.boot.CommandLineRunner* interface.

b. Annotate the class with **@Controller** so that Spring can detect it.

c. Define a field of type *org.apache.log4j.Logger*:

```
private Logger log = Logger.getLogger(ShippingController.class);
```

d. Define a field of type *Shipping* and auto-wire it:

```
@Autowired
private Shipping shipping;
```

@Autowired is a Spring annotation that injects a reference using the type of the field, in this case, *Shipping*. Since you wrote the *ShippingImpl* class to implement that interface and annotated it with @Component, Spring will inject its reference here.

e. Complete the *run()* method so that it uses the shipping service to print the domestic and international shipping rates to the logger at *info()* level using zip and country of your choice.

Hint: Call log.info() to output to the logger at *info* level.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

_8. Run the program:

At the command prompt, enter:

```
gradlew bootRun
```

You should see your program's output intermingled with diagnostics from Spring boot.

_9. Get rid of the Spring Boot banner and run from within Eclipse:

a. Modify the *main()* method in DemoApplication so it looks like:

```
//        SpringApplication.run(DemoApplication.class, args);
      SpringApplication app = new SpringApplication(DemoApplication.class);
      app.setBannerMode(Mode.OFF);
      app.run(args);
```

b. In the Eclipse Package Explorer, right-click on DemoApplication.java and choose *Run As - Java Application* - you should see the output in the Eclipse Console view.

Part 2: A Spring Boot Web Application

In this part, you will create a simple Web application using Spring Boot.

Steps:

_1. In your command prompt, use the Spring Boot CLI to generate a Web application:

a. First, change to the new project's directory:

```
cd ..\lab07p2
```

b. Use the Spring Boot CLI and Initialzr to generate a starter project:

```
spring init -b 1.3.1.RELEASE -d=web --build gradle -x
```

The CLI will contact the start.spring.io and then generate a project and unzip it. Use Windows Explorer to examine the project structure, comparing to the picture in the Intro to Spring Boot chapter in the course notes. It should be similar, but with different names.

_2. In the same fashion as in earlier labs, in the command prompt, generate an Eclipse project.

_3. In the same fashion as in earlier labs, import the existing project into Eclipse.

_4. Create the view pages for the application:

a. In the project's *src/main/resources/static* folder, create a file named **segment.json**, and in it, write the JSON for an object containing Segment data:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
{
   "segmentId":       46765,
   "segmentDate":     "2017-02-18",
   "flightNumber":    3462,
   "originatingCity": "ORD",
   "miles":           1362
}
```

b. In the *src/main/resources/static* folder, create an HTML file named **view-segment.html**, and in it, write a React page that uses axios to request *http://localhost:8080/segment.json* and displays the Segment in a React component.

Note that you can copy and paste some of the React code from the *lab05part1* project and some of the AJAX stuff from the *lab05part2* project. Also note that since there's only one Segment, you won't need to iterate.

_5. Test the application:

a. In the same fashion as in the first part of lab, in your command prompt, use *gradlew bootRun* to start an embedded Tomcat instance and your application. Leave the command prompt running.

b. Open a browser and enter this URL:

```
http://localhost:8080/view-segment.html
```

You should see your application's Segment display.

c. You can press Ctrl+C on the command prompt where you executed bootRun to kill the embedded Tomcat instance.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 8: Spring Annotations

In this lab, you will work with configuring Spring beans using annotations.

Objectives:

- To configure Spring beans using annotations

Part 1: Getting Started

Steps:

_1. Your first job is to generate an Eclipse project for this lab. We will use Spring Boot and the Gradle build tool to satisfy dependencies and generate the project. Follow this procedure:

   a. In the Windows Start menu, at the bottom, enter **cmd** and press Enter to open a command prompt window.

   b. Change to the lab's directory. If your computer has the standard lab setup:

   ```
   cd \Users\username\webclass\workspace\lab08
   ```

   c. Use Spring Boot to generate a starter project:

   ```
   spring init -b 1.3.1.RELEASE  --build gradle -x
   ```

   d. Open the Gradle build script and see if you can make any sense of it:

   ```
   write build.gradle
   ```

   Note that it specifies the *eclipse* plugin that knows how to generate Eclipse projects. Close WordPad when you are finished examining the file (you don't need to modify it).

   e. Generate an Eclipse project and download the dependencies:

   ```
   gradlew eclipse
   ```

_2. Your next job is to import the project into Eclipse. Follow this procedure:

   a. From the Eclipse menu, choose *File - Import* to start the wizard.

   b. On the first wizard page, expand the *General* category and highlight *Existing projects into workspace*, then press Next.

   c. On the next wizard page, click the Browse button next to the *Select root directory* button - Eclipse will show the *workspace* folder. Just click *Open* to select that folder.

   d. In the project list, ensure that ONLY *lab08* is selected. Then look below the project list and ensure that the *Copy projects into workspace* checkbox is NOT selected. Press Finish to import the project.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

e. In the Project Explorer, expand the project and note that it contains the build.gradle script and that the Referenced Libraries folder contains several JARs.

f. In the *src/main/java* folder, right-click on *com.example.demo* and choose *Refactor - Rename* and rename the package to **com.bigbank**.

Part 2: Using Annotations for Dependency Injection

In this part, you will create the Spring beans for a hypothetical bank's loan-processing service. You will create classes named **Loan**, **Borrower** and an interface named **LoanService** with an implementation class named **LoanServiceImpl**. You will then use annotations to connect the beans together.

Steps:

_1. In the *src/main/java* folder, create a class named **Loan** in the *com.bigbank.domain* package. Complete the class:

a. Define fields:

```
private int loanID;
private int amount;
private int purchasePrice;
private String status;
private String propertyAddress;
```

b. Generate get/set methods for the fields.

c. Annotate the component so it's a Spring bean named **loan01**.

d. Use annotations to initialize the fields to values of your choice. The loan amount should be less than $30,000. You don't need to initialize the status, since it will be set when the program runs.

_2. Create another class in the *com.bigbank.domain* package named Borrower. Complete the class:

a. Annotate the class so it's a Spring bean named **borrower01**.

b. Write fields:

```
private int borrowerID;
private String name;
private String address;
private String phone;
private Loan theLoan;
```

c. Generate get/set methods for the fields.

d. Annotate the *theLoan* field to inject a reference to **loan01** by name.

e. Use annotations to initialize the other fields to values of your choice.

_3. Create an interface named **LoanService** in the *com.bigbank.services* package. Complete the interface by writing a single method:

```
boolean processLoan(Borrower b);
```


Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

_4. Create a class named **LoanServiceImpl** in the *com.bigbank.services* package that implements the LoanService interface. Complete the processLoan method:

a. Annotate the class so it's a Spring component. It doesn't need a name since you will inject it by type.

b. Calculate a random number that we will use to simulate a loan threshold:

```
Random rand = new Random();
int threshold = rand.nextInt(30000);
```

c. Implement the logic of this method:

- If the requested loan amount is more than the purchase price, the loan is "Rejected".
- If the requested loan amount is greater than the threshold, the loan is "Rejected".
- Otherwise, the loan is "Approved".

d. If the loan is approved, set its status to **Approved**. Otherwise set it to **Rejected**.

_5. Create a new class named **LoanManager** in the *com.bigbank.domain* package. Complete the class:

a. Annotate the class so it's a component named **loanManager**.

b. Define a field that references the LoanService:

```
private LoanService loanService;
```

c. Annotate the field to inject a reference to the LoanServiceImpl by type.

d. Write a method that uses the service:

```
public String applyForLoan(Borrower b)
{
    String retVal = "";

    boolean rating = loanService.processLoan(b);

    retVal = "Loan status is " +
        b.getTheLoan().getStatus() + ".";

    if (rating)
    {
        retVal += " Here's a free toaster!";
    }
    else
    {
        retVal += " Sorry, no loan today";
    }

    return retVal;
}
```

_6. Create a new class named **LoanApplicationController** in the *com.bigbank.controllers* that implements the *org.springframework.boot.CommandLineRunner* interface, then modify it:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

a. Annotate the class with **@Controller** so that Spring can detect it.

b. Define a field of type *org.apache.log4j.Logger*:

```
private Logger log = Logger.getLogger(LoanApplicationController.class);
```

c. Inject a reference to the *borrower01* bean, wiring by name (use @Resource).

d. Inject a reference to the LoanManager bean, wiring by type (use @Autowired).

e. Within the *run()* method, call the LoanManager's applyForLoan() method and log the result at *info* level.

_7. In your command prompt, use *gradlew bootRun* to execute the program. You should see the loan-application outcome.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 9: Spring Java Configuration

In this lab, you will work with configuring Spring beans using Java configuration.

Objectives:

- To configure Spring beans using Java configuration

Part 1: Getting Started

To give you a running start, you will copy the last lab. If you didn't finish the last lab, either finish it before starting this lab or ask your instructor for help obtaining the solution.

Steps:

_1. In Eclipse, in the Package Explorer, right-click on *lab08* and choose *Copy*. Then right-click on a blank area in the Package Explorer and choose *Paste*, naming the copied project **lab09**.

_2. Clean up the copied project so you have a clean slate for Java configuration:

   a. In the *lab09* project's *com.bigbank.domain*, edit Borrower and LoanManager and remove all of the Spring annotations. Leave the Loan class as is - you will use annotations to configure this class.

      You can use the *Source - Organize Imports* menu item to remove unneeded imports.

      In LoanManager, generate get/set methods for the *loanService* field so you can later use "setter" injection to initialize it.

   b. In the *com.bigbank.service* package, edit the LoanServiceImpl class and remove all Spring annotations.

   c. Copy the *spring.xml* file from *lab06* into the new projects *src/main/resources* folder.

      Then open the spring.xml file and delete the *context:component-scan* element.

_3. Open a command prompt and change to the *lab09* folder, then try running the project with *gradlew bootRun* - you should get a NoSuchBeanDefinitionException.

Part 2: Using Java Configuration

In this part, you will configure the project's beans using a mixture of Java configuration, annotations and XML. Note that while this is not necessarily a real-world approach, it will give you practice using the various techniques.

Steps:

_1. Examine the *com.bigbank.domain.Loan* class and note that it uses annotations to configure a Spring bean named *loan01*.

_2. Edit spring.xml and define a bean of class *com.bigbank.domain.Borrower* named **borrower01** using values of your choice for the simple properties and a reference to *loan01* for the *theLoan* property.

_3. For the application to work, you need to configure two more beans - you will use Java configuration for those. Follow this procedure:

   a. In *src/main/java*, create a new class named **LoanManagerConfig** in the *com.bigbank.configuration* package.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Annotate the class with @Configuration.

b. Annotate the class with @ComponentScan with the *com.bigbank.domain* package - this will bring in the *loan01* bean into the configuration.

c. Annotate the class with @ImportResource to reference the *spring.xml* file on the CLASSPATH. This will bring in the *borrower01* bean into the configuration.

d. Within the class, define a bean-creation method that creates a bean of type LoanServiceImpl. The method's skeleton should look like:

```
@Bean
public LoanService loanService()
{

}
```

e. Then write another bean-creation method that creates and returns a configured LoanManager bean named **loanManager**. Be sure to inject the reference to the LoanService you defined in the previous method.

_4. Run the program again. It should work the same as it did in the last lab, but now the bean configuration is scattered across XML, annotation-based and Java configuration.

Optional Challenge: Using the Groovy DSL

In this part, you will modify the program so it defines another bean using the Groovy DSL. Since this is a challenge lab, we will give you minimal instructions.

Steps:

_1. Use the Groovy DSL to configure another Loan bean named **loan02**. You will need to update the build.gradle script, run it, refresh the project in Eclipse, write a spring.groovy configuration file and update LoanManagerConfig to reference the Groovy configuration file. Put spring.groovy in the *src/main/resources* folder.

Then add code to LoanApplicationController to inject the loan02 bean and log its properties.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 10: This Lab Intentionally Left Blank

In order to prevent all life as you know it stopping instantaneously, and every molecule in your body exploding at the speed of light, this lab is blank.

Objectives:

- To prevent Total Protonic Reversal

Steps:

_1. DON'T cross the streams!

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 11: This Lab Intentionally Left Blank

Due to the inherent unfairness of the universe, this lab is intentionally left blank.

Objectives:

- To prevent existential angst

Steps:

_1. Scar: Life's not fair, is it? You see, I... well, I shall never be king. And you... shall never see the light of another day.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 12: Spring and the Web

In this lab, you will put a Web front end on a Spring application.

Objectives:

- To use SpringMVC

Part 1: Spring MVC with Java Configuration

In this part, you will write a SpringMVC Web application that will act as the user interface for your Spring beans. You will use the Java configuration approach.

Steps:

_1. Your first job is to generate an Eclipse project for this lab. We will use the Spring Boot and Gradle to satisfy dependencies and generate the project. Follow this procedure:

   a. In the Windows Start menu, at the bottom, enter **cmd** and press Enter to open a command prompt window.

   b. Change to the lab's directory. If your computer has the standard lab setup:

```
cd \Users\username\webclass\workspace\lab12
```

   c. Use the Spring Boot Initializr to generate a starter project:

```
spring init -b 1.3.1.RELEASE -d=web --build gradle -x
```

   d. In the same fashion as in earlier labs, use the *gradlew* script to generate an Eclipse project, then import it into Eclipse.

   e. Expand the new project's *src/main/java* folder and right-click on the *com.example.demo* package and choose *Refactor - Rename* to change the package name to **com.oaktreeair**.

   f. In the *com.oaktreeair* package, right-click on DemoApplication.java and rename it to **FrequentFlierProgram**.

_2. Next, copy files from the *lab06JavaConfig* project to the new Web project:

   a. In the Project Explorer pane, expand the *lab06JavaConfig* project's *src/main/java* folder, then right-click on the *com.oaktreeair.ffprogram* package and choose *Copy*.

   b. In the Project Explorer, expand the *lab12* folder, then right-click on the *src/main/java* folder and choose *Paste* to copy the Spring bean classes to the new project.

_3. Next, let's create a simple Spring Java configuration file that defines a couple of Flier components (note that this is artificial just for this lab: in the real world, the Flier info would be stored in a database):

   a. Right-click on the *com.oaktreeair* package and create a class named **FrequentFlierConfig**.

   b. Annotate the class with *@Configuration*.

   c. Write two methods named **flier01()** and **flier02** that create, initialize and return Flier objects. Annotate each of the methods with *@Bean*. Initialize the Flier objects to values of your choice. Note that the method

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

names assign names to the beans, so when the program runs, the container will have beans named *flier01* and *flier02*.

Note: Be sure to assign each of the Flier objects an initialized ContactInfo object.

_4. Now, create an HTML input form so the user can enter a flier bean ID:

a. Right-click on the project and create a new folder named **src/main/webapp/WEB-INF/jsp**.

b. Right-click on the newly created folder and choose *New - Other - Web - JSP File* and create a JSP named **findFlier.jsp**. On the third page of the wizard, uncheck the box to "Use JSP Template".

Eclipse opens the new JSP into the editor.

c. Enter the following:

```
<html>
  <head>
    <title>Find Flier by Bean ID</title>
  </head>
  <body>
    <form action="findFlier.html" method="post">
      <p>
        Enter flier bean ID:
        <input type="text" name="flierId">
      </p>
      <p>
        <input type="submit" value="Find">
      </p>
    </form>
  </body>
</html>
```

This defines a standard HTML input form into which the user can enter a bean ID.

_5. Next, define a JSP that will display the results of finding a bean:

a. Right-click on the *jsp* folder and create a new JSP named **showFlier.jsp**, again using no JSP Template.

b. Enter the following text:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core"
  prefix="c"%>
<html>
  <head>
    <title>Flier Information</title>
  </head>
  <body>
    <h3>Flier Information</h3>
    <p>${msg}</p>
    <p>
      <c:if test="${not empty flierName}">
        Name: ${flierName}
      </c:if>
    </p>
    <p>
      <c:if test="${not empty flierEmail}">
        Email: ${flierEmail}
      </c:if>
    </p>
  </body>
</html>
```

This page looks for the flier information stored at request scope and displays it.

Note: You can ignore errors and warnings in this file on the JSTL *taglib* directive and c:if lines.

_6. Next, create an *application controller* class that responds to the input form and stores the flier info in a *model and view* object:

a. Create a new class in a new *com.oaktreeair.ffprogram.controllers* package named **FindFlierController**.

b. Configure the controller class with the following annotations:

```
@Controller
@RequestMapping("/findFlier")
```

These annotations configure the class as a SpringMVC controller and indicate that it should respond to URLs that contain "/findFlier".

c. Within the class, setup an injection into a field so that Spring can inject an application-context reference:

```
 @Autowired private ApplicationContext applicationContext;
```

d. Write a method that Spring MVC will invoke when the user issues an HTTP "get" to the controller:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
@RequestMapping(method = RequestMethod.GET)
public String setupForm(Model model)
{
    return "findFlier";
}
```

Note that the method does nothing yet; you will add to it later.

e. Start writing a method to handle the posted form submit request:

```
@RequestMapping(method = RequestMethod.POST)
public String submitForm(@RequestParam("flierId") String flierId,
        Model model)
{


    return "showFlier";
}
```

Note how we annotate one of the method's parameters with @RequestParam so that Spring MVC injects the value that the user enters into the form. Check back in findFlier.jsp and ensure you understand how this will work.

f. In the *submitForm* method, write a try-catch block:

```
try
{

}
catch(NoSuchBeanDefinitionException e)
{

}
```

g. In the *try* block, use the Spring application context in the normal way to attempt to get the bean whose bean ID is stored in the flierId parameter:

```
Flier flier = . . .;
```

This will throw a NoSuchBeanDefinitionException if there's no bean with the specified ID in the application context.

Then store the flier info into the model object:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
model.addAttribute("msg", "Flier found successfully");
model.addAttribute("flierName", flier.getFlierName());
ContactInfo inf = flier.getContactInfo();
model.addAttribute("flierEmail", inf.getEmailAddress());
```

h. In the *catch* block, store a not-found message into the model-and-view object:

```
model.addAttribute("msg", "Flier " + flierId +
        " not found");
```

_7. Edit the *src/main/resources/application.properties* file and add:

```
spring.mvc.view.prefix: /WEB-INF/jsp/
spring.mvc.view.suffix: .jsp
```

This configures Spring MVC so it can find your JSPs.

Note: Ensure that there are no trailing spaces in those two lines.

_8. Edit the build.gradle file and modify it:

a. In the *apply-plugin* list, add:

```
apply plugin: 'war'
```

b. In the bottom-most *dependencies* block, add:

```
providedRuntime("org.apache.tomcat.embed:tomcat-embed-jasper")
providedRuntime("javax.servlet:jstl")
```

_9. To run and test, in your command prompt, use *gradlew bootRun* to start the application within the embedded Tomcat server. Then open a web browser and enter:

```
http://localhost:8080/findFlier
```

You should see your input form. Enter **flier01** into the form and press Submit - you should see the flier info.

Press the back button in the browser and enter **flier04** and press Submit - you should see the not-found message.

_10. The application worked, but let's fix up a minor inconvenience:

a. Open findFlier.jsp into the editor and find the *input* tag for the flier ID. Add the following attribute to the tag:

```
value="${flierId}"
```

b. Open the FindFlierController.java class into the editor, and modify the setupForm method so that it initializes a flier ID for the form:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
        model.addAttribute("flierId", "flier01");
```

_11. To test with the updates, in your command prompt, kill, then restart the application.

Back your web browser, do a hard refresh (hold the Ctrl key and press Refresh) of the *http://localhost/findFlier* URL and note that the form is "pre-filled" with a good flier bean ID -- that's because your controller returned a model object from the *setupForm* method.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 13: Spring and JDBC

In this lab, you will write an application that uses Spring's SimpleJdbcTemplate support for writing data access objects (DAO).

Objectives:

- To access relational data using Spring SimpleJdbcTemplate

Part 1: Setting Up the Environment

In this part, you will configure an H2 database that your Spring program will work with.

Steps:

_1. Next, you need to start the H2 database server and its adminstrative GUI, which is Web-based:

Using Windows Explorer or My Computer, navigate to **{Lab Installation Directory}/lib/h2/h2-1.3.161.jar** and double-click on the JAR - this starts the database running and opens a Web-based administration GUI.

_2. In the H2 GUI, set the *JDBC URL* to **jdbc:h2:tcp://localhost/~/lab13**, typing this VERY carefully.

Set both the *User Name* and *Password* to **sa**, then press Connect.

_3. Enter the following into the large entry box and then press the Run (Ctrl+Enter) button to create and populate your database (you can copy and paste from the starters.txt file in **{Lab Installation Directory}/starters/spring/lab13}**):

```
CREATE TABLE Segment
(
   SegmentNumber   INTEGER NOT NULL IDENTITY,
   SegmentDate     DATE,
   FlightNumber    INTEGER,
   OrigCity        VARCHAR(10),
   Miles           INTEGER,
   CONSTRAINT  PK_Item  PRIMARY KEY(SegmentNumber)
);


INSERT INTO Segment VALUES (NULL, '2009-10-18', 333, 'SFO', 367);
INSERT INTO Segment VALUES (NULL, '2009-10-18', 745, 'LAX', 1900);
INSERT INTO Segment VALUES (NULL, '2009-10-22', 453, 'BOS', 1900);
INSERT INTO Segment VALUES (NULL, '2009-10-18', 112, 'LAX', 367);
```

Press the Clear button then enter and run:

```
SELECT * FROM segment
```

You should see four segment rows.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Part 2: Setting Up the Web Project

In this part, you will create a Spring MVC application that will act as the user interface for a Spring application that queries relational data using Spring's SimpleJdbcTemplate.

Steps:

_1. This lab depends on successfully completing the basic parts of *lab06* - it doesn't depend on the "experiments". If you did not finish the basic part of lab06, you should either finish it or ask the instructor to help you get that lab's solution.

_2. In the same fashion as in earlier labs, open a command prompt and change to **{Lab Installation Directory}/workspace/lab13**. Using the Spring Boot Initialzr to generate a project with *web*, *jdbc* and *h2* dependencies:

```
spring init -b 1.3.1.RELEASE -d=web,jdbc,h2 --build gradle -x
```

Run the *gradlew* script to generate an Eclipse project, then import the project into Eclipse. **Note:** Be sure to only import lab13!

_3. After importing the project into Eclipse, continue configuring the project:

a. Expand the new project's *src/main/java* folder and right-click on the *com.example.demo* package and choose *Refactor - Rename* to change the package name to **com.oaktreeair**.

b. In the *com.oaktreeair* package, right-click on DemoApplication.java and rename it to **FrequentFlierProgram**.

_4. Next, copy files from the *lab06JavaConfig* project to the new Web project:

a. In the Project Explorer pane, expand the *lab06JavaConfig* project's *src/main/java* folder, then right-click on the *com.oaktreeair.ffprogram* package and choose *Copy*.

b. In the Project Explorer, expand the *lab13* folder, then right-click on the *src/main/java* folder and choose *Paste* to copy the Spring bean classes to the new project.

_5. Edit *src/main/resources/application.properties* and add the following lines to provide the database configuration so that your application can use the H2 database:

```
spring.datasource.url=jdbc:h2:tcp://localhost/~/lab13
spring.datasource.username=sa
spring.datasource.password=sa
spring.datasource.driver-class-name=org.h2.Driver
```

Note: Please ensure that there are no extra spaces at the end of any of these lines.

Part 3: Spring JdbcTemplate DAO

In this part, you will create a data access object (DAO) class using the Spring JdbcTemplate type.

Steps:

_1. Create an interface for your DAO:

a. In the Project Explorer, right-click on the *src/main/java* folder and choose *New - Interface* and create an interface named **SegmentDao** in a new package named **com.oaktreeair.ffprogram.dao**.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

b. Add the following methods to the interface:

```
public int getSegmentCount();
public Collection<Segment> findAllSegments();
public int insertSegment(Segment s);
```

You will need to import *java.util.Collection* and *com.oaktreeair.ffprogram.Segment*.

_2. Create the DAO implementation class:

a. In the Project Explorer, right-click on the *com.oaktreeair.ffprogram.dao* package and choose *New - Class* to start the wizard.

b. On the first wizard page, ensure that the *Package* is **com.oaktreeair.ffprogram.dao**.

For the *Name*, enter **SegmentDaoImpl**.

For the *Interfaces*, press the Add button, and in the *Choose interfaces* box, start typing **Seg**, then select *SegmentDao*.

Press OK followed by Finish to complete the wizard. Eclipse opens the new class into the editor.

_3. Annotate the DAO as a repository component and give it a name:

```
@Repository("segmentDao")
```

Import the correct type. Note the Spring ID of the DAO is *segmentDao*.

_4. Let's start by using dependency injection to obtain the resources the DAO needs:

a. Define a field to hold the JDBC template:

```
private JdbcTemplate template;
```

Import the correct type.

b. Inject a data-source reference and create the template:

```
@Autowired
public void setDataSource(DataSource ds)
{
    template = new JdbcTemplate(ds);
}
```

Import the javax.sql.DataSource type.

_5. Let's start by implementing only the *getSegmentCount* method:

a. Execute a SQL command via the template to retrieve the count of rows in the Segment table:

```
int count = template.queryForObject(
        "SELECT COUNT(*) FROM Segment", Integer.class);
```

b. Modify the *return* statement to return the count.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Part 4: Web Front End for the DAO

In this part, you will configure the DAO with Spring and write a simple controller to invoke its *getSegmentCount* method.

Steps:

_1. Next, create a simple controller that will access the segment count via the DAO:

   a. Create a class named **SegmentController** in a new packaged named **com.oaktreeair.ffprogram.controllers**.

   b. Annotate the class as a SpringMVC controller.

   c. Autowire a reference to the SegmentDao interface.

   d. Write a method that returns a String and accepts a Spring MVC Model. Annotate the method so it responds to a GET request, with a path that ends in */segmentcount*. The method should call the DAO's *getSegmentCount()* method and store the count in the Model. Return "displaysegmentcount" as the name of the JSP that will run after the controller.

_2. Right-click on the project and create a new folder named **src/main/webapp/WEB-INF/jsp**.

   In that new directory, create a JSP named **displaysegmentcount.jsp** that displays the count from the Model.

_3. Edit the *src/main/resources/application.properties* file and add:

```
spring.mvc.view.prefix: /WEB-INF/jsp/
spring.mvc.view.suffix: .jsp
```

   This configures Spring MVC so it can find your JSPs.

   Note: Ensure that there are no trailing spaces in those two lines.

_4. Edit the build.gradle file and modify it:

   a. In the *apply-plugin* list, add:

```
apply plugin: 'war'
```

   b. In the bottom-most *dependencies* block, add:

```
providedRuntime("org.apache.tomcat.embed:tomcat-embed-jasper")
providedRuntime("javax.servlet:jstl")
```

_5. To run and test, in your command prompt, use *gradlew bootRun* to start the application within the embedded Tomcat server. Then open a web browser and enter:

```
http://localhost:8080/segmentcount
```

   You should see the segment count (4).

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Part 5: Completing the Application

In this part, you will complete the DAO and write additional controller function and JSPs as the user interface.

Steps:

_1. You will start by implementing the DAO's *findAllSegments* method.

First, create a *nested inner class* that knows how to create Segment objects from rows in the Segment database table:

a. In the Project Explorer, right-click on *SegmentDaoImpl* and choose *New - Class* to start the wizard.

b. On the first wizard page, ensure that the *Package* is *com.oaktreeair.ffprogram.dao*.

Put a checkmark in the *Enclosing type* checkbox - this causes the wizard to generate an inner class.

For the *Name*, enter **SegmentRowMapper**.

For *Interfaces*, press the Add button and start typing **RowMapper** and then select *RowMapper* from the Spring framework and press OK followed by Finish to complete the wizard.

_2. The wizard gave you a head start, but you need to tweak the inner class a bit:

a. Edit the inner class definition so it looks like:

```
public class SegmentRowMapper
  implements RowMapper<Segment>
```

b. Inside the nested class, add the *mapRows* method definition:

```
public Segment mapRow(ResultSet rs, int rowNum)
   throws SQLException
{
}
```

_3. Complete the SegmentRowMapper inner class's *mapRow* method:

a. Create a new Segment object using its zero-argument constructor.

b. Call each of the "set" methods on the Segment object, initializing with data from the result set. For example, to set the Segment's *flightNumber* property:

```
seg.setFlightNumber(rs.getInt("FlightNumber"));
```

Be sure to call ALL of the "set" methods. For your convenience, here is the Segment table schema:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
CREATE TABLE Segment
(
   SegmentNumber   INTEGER,
   SegmentDate     DATE,
   FlightNumber    INTEGER,
   OrigCity        VARCHAR(10),
   Miles           INTEGER
)
```

Note that you will need to map SQL types to their equivalent Java types by calling the appropriate getXXXX() method on the result set.

c. Return the fully initialized Segment object.

_4. Update the DAO class to create a row mapper object:

a. In the SegmentDaoImpl class (NOT the SegmentRowMapper inner class), define a field to hold an instance of the SegmentRowMapper class:

```
private SegmentRowMapper mapper;
```

b. Modify the *setDataSource()* method so that it creates an instance of the SegmentRowMapper class using the zero-argument constructor, storing the reference in the field you just defined.

_5. Now complete the *findAllSegments* method:

a. Call the template's *query* method to return a list of Segments, using the mapper to convert from result set to Segment object:

```
List<Segment> segments = template.query(
        "SELECT * FROM Segment", mapper);
```

b. Modify the *return* statement to return the list.

_6. Update the SegmentController to have a method that responds to a GET request to the path */allsegments*. This method should store the segment list in the Model for display by a new JSP. For extra credit (!), display the list in an HTML table using JSTL. Here's some JSTL to help you get started (assuming you named the model attribute *segments*):

```
<c:forEach items="${segments}" var="aSeg">
  <tr>
    <td>${aSeg.segmentNumber}</td>
      . . .
  </tr>
</c:forEach>
```

_7. Run and test.

_8. If you have time, implement the DAO's *insertSegment* method. Then write an HTML input form to let the user enter all of the Segment data except the segmentNumber (that's an auto-generated column in the database). The

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

HTML form's Submit button should invoke a new controller method that retrieves the HTML form data and calls the DAO's *insertSegment* method to insert the new segment into the database.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Optional Lab 14: Spring Transactions

In this lab, you will write a program that uses Spring declarative transactions.

Objectives:

- To use Spring declarative transaction demarcation

Part 1: Setup

In this part, you will update and import an Eclipse project that contains a starting point for the lab.

Steps:

_1. Your first job is to update the starter project's dependencies:

   a. In the Windows Start menu, at the bottom, enter **cmd** and press Enter to open a command prompt window.

   b. Change to the lab's directory. If your computer has the standard lab setup:

   ```
   cd \Users\username\webclass\workspace\lab14
   ```

   c. Generate an Eclipse project and download the dependencies:

   ```
   gradlew clean eclipse
   ```

_2. As you have done in earlier labs, import the project into Eclipse.

_3. Examine the new project looking in the *com.oaktreeair.ffprogram.domain* and *com.oaktreeair.ffprogram.dao* packages. Figure 1 summarizes the application:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

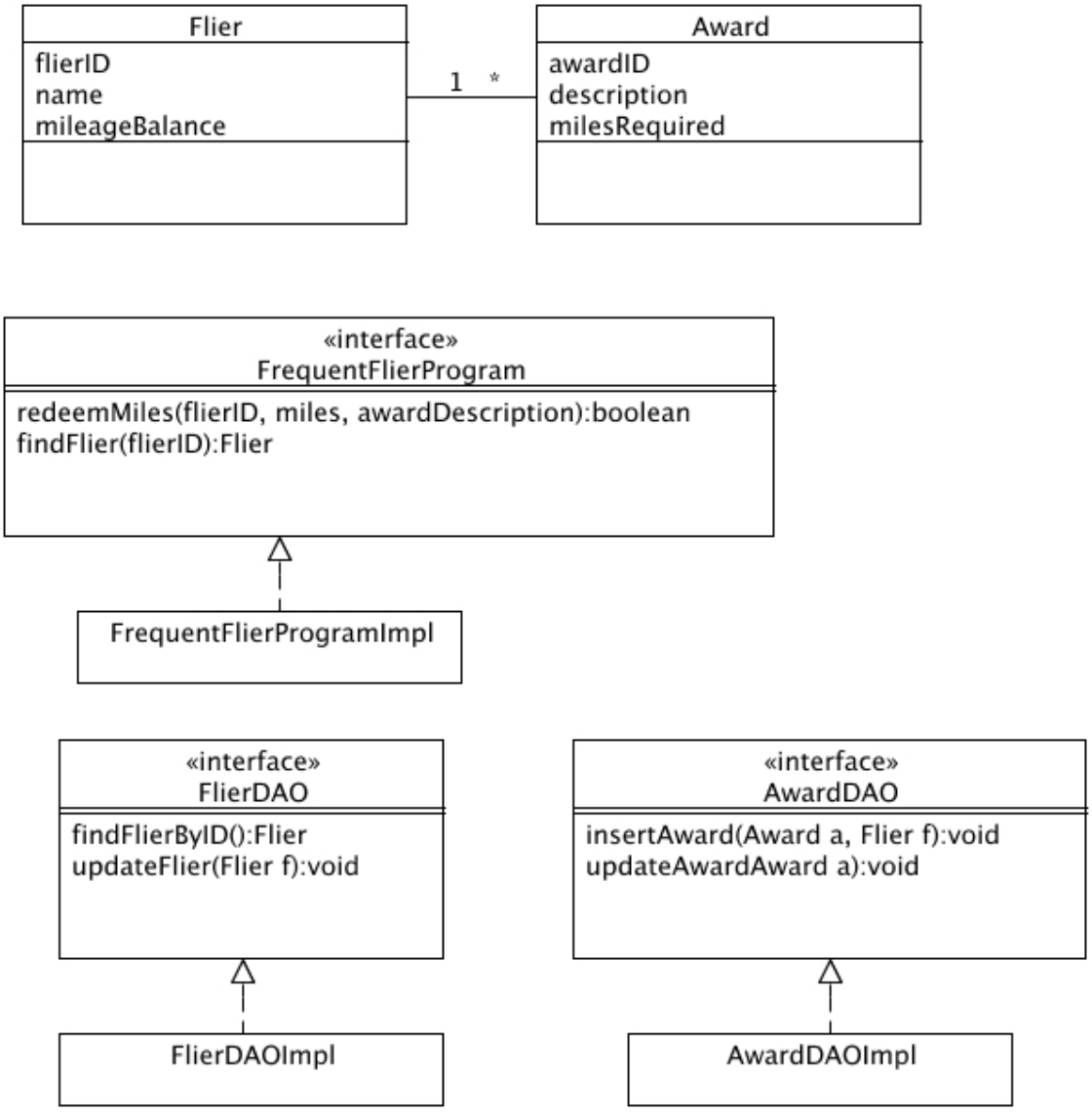


Figure 1: Application Model

We have provided you will all of the interfaces; you will write the implementation classes. In addition, we have provided you with the Flier.java and Award.java classes.

_4. Before you can run the Web application, you need to create the database tables and populate them with sample data. We have provided you with an SQL script for this purpose; follow these steps to run the script:

a. Look in your computer's Windows System Tray (lower right corner of the screen) for a square, yellow icon that represents the H2 Database Engine (hover your mouse over it to see its title). If you see it, right-click on the icon and choose *Exit*, then close any browser windows for the H2 database.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

b. Using Windows Explorer or My Computer, navigate to **{Lab Installation Directory}/lib/h2/h2-1.3.161.jar** and double-click on the JAR - this starts the database running and opens a Web-based administration GUI.

c. In the admin console, CAREFULLY change the JDBC URL to **jdbc:h2:tcp://localhost/~/lab14**, then enter **sa** for both username and password, then press Connect.

d. Back in Eclipse, look in the project's root directory, then double-click on *createfliers.sql* to open the script into the editor. Examine this large script and see if you can make sense of it, then press Ctrl+A to select all of its text, then Ctrl+C to copy it into the clipboard.

e. Back in the H2 Console, paste the contents of the clipboard into the large entry field.

To run the script, press the *Run (Ctrl+Enter)* button. The script should execute with no errors.

Once the script completes, you should see the *AWARD* and *FLIER* tables listed in the left-hand pane of the H2 Console.

f. To ensure that there's good data in the tables, press Clear, then enter **SELECT * FROM Flier** and press the *Run* button. You should see two rows of Flier information.

g. You can leave the H2 Console window up so that you can explore the database whenever you like, but for now, go back to Eclipse.

Part 2: Writing the DAOs

In this part, you will write Spring JDBC DAOs that will be accessed from the Spring object model. We have provided you with the DAO interfaces: your job is to write the implementations.

Steps:

_1. Create a new class named **AwardDaoImpl** in the *com.oaktreeair.ffprogram.dao* package that implements the *AwardDao* interface.

Complete the DAO:

a. Annotate the class with **@Repository**, assigning the name **awardDao**.

b. Annotate each of the methods the class so they participate in Spring transactions:

```
@Transactional(propagation=Propagation.MANDATORY)
```

c. Define a field, autowiring it to reference the FlierDao:

```
@Autowired
private FlierDao flierDao;
```

d. As you did in an earlier lab's SegmentDaoImpl, create an **AwardRowMapper** inner class that creates Award objects from a ResultSet. You can look in the createfliers.sql script for the table schema.

For the *flierID* foreign-key, you can use the FlierDao (you will complete in a moment) to fetch the associated Flier object:

```
long flierID = rs.getLong("FlierID");
a.setFlier(flierDao.findFlierByID(flierID));
```


Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

e. In the AwardDaoImpl class, NOT the inner class, define these fields:

```
private JdbcTemplate template;
private AwardRowMapper mapper;
private DataSource theDataSource;
```

f. Write a method to initialize those fields - look in the *application.resources* properties file for the data-source configuration:

```
@Autowired
public void setDataSource(DataSource ds)
{
    theDataSource = ds;
    template = new JdbcTemplate(ds);
    mapper = new AwardRowMapper();
}
```

g. Implement the *insertAward* method:

```
SimpleJdbcInsert inserter = new SimpleJdbcInsert(theDataSource);
inserter.setTableName("award");
inserter.usingGeneratedKeyColumns("AwardID");
Map<String, Object> parms = new HashMap<String, Object>();
parms.put("Description", award.getDescription());
parms.put("MilesRequired", award.getMilesRequired());
parms.put("FlierID", flier.getFlierID());

Number id = inserter.executeAndReturnKey(parms);

award.setAwardID(id.longValue());
flier.getAwards().add(award);

return id.intValue();
```

h. Implement the *updateAward* method:

```
template.update(
  "UPDATE Award SET Description=?, MilesRequired=?, FlierID=?",
     award.getDescription(), award.getMilesRequired(),
     award.getFlier().getFlierID());
```

i. Implement the *findAwardsForFlier* method:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
Collection<Award> awards = template.query(
    "SELECT * FROM Award WHERE flierID=?",
    mapper, flierID);

return awards;
```

_2. Create a new class named **FlierDaoImpl** in the *com.oaktreeair.ffprogram.dao* package that implements the *FlierDao* interface.

Complete the DAO:

a. Annotate the class with **@Repository**, assigning the name **flierDao**.

b. Annotate each of the methods the class so they participate in Spring transactions:

```
@Transactional(propagation=Propagation.MANDATORY)
```

c. As you did in the last part, create an **FlierRowMapper** inner class that creates Flier objects from a ResultSet. You can look in the createfliers.sql script for the table schema.

boolean retVal = false; Long idFlier = new Long(flierID); Flier flier = flierDAO.findFlierByID(idFlier); int balance = flier.getMileageBalance(); if (balance - miles >= 0) { // create a new award and add // to flier's award collection Award award = new Award(); award.setMilesRequired(miles); award.setDescription(awardDescription); awardDAO.insertAward(flier, award); // reduce flier's balance flier.setMileageBalance(balance - miles); flierDAO.updateFlier(flier); retVal = true; } return retVal; You don't need to populate the flier's Award collection.

d. In the FlierDaoImpl class, NOT the inner class, define these fields:

```
private JdbcTemplate template;
private FlierRowMapper mapper;
```

e. Write a method to initialize those fields - look in application.resources to see the data-source configuration:

```
@Autowired
public void setDataSource(DataSource ds)
{
    template = new JdbcTemplate(ds);
    mapper = new FlierRowMapper();
}
```

f. Implement the *findFlierByID* using the example in the course notes as a guide.

g. Implement the *updateFlier* method in the same fashion as you did for the similar method in AwardDaoImpl. Note that you don't need to do anything with the flier's award collection in this method.

_3. Next, write the FrequentFlierProgramImpl class:

a. Create a new class named **FrequentFlierProgramImpl** in the *com.oaktreeair.ffprogram.domain* package that implements the *FrequentFlierProgram* interface.

b. Annotate the class with **@Service**, assigning a name of **frequentFlierProgram**.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

c. Annotate the class so all of its methods are transactional with *REQUIRED* propagation.

d. Define fields for the DAOs:

```
private FlierDao flierDAO;
private AwardDao awardDAO;
```

Annotate both fields with **@Autowired** so that Spring injects the DAO references.

e. Implement the *findFlierByID* method:

```
Long id = new Long(flierID);

Flier theFlier = flierDAO.findFlierByID(id);
Collection<Award> awards = awardDAO.findAwardsForFlier(id);
theFlier.setAwards(awards);

return theFlier;
```

f. Implement the *redeemMiles* method:

g. Save FrequentFlierProgramImpl.java.

_4. Next, take a look at the provided application.resources properties file and note that it configures data source properties using a JDBC URL and other properties. Since we are using Spring Boot, it will autoconfigure a DriverManagerData data source (recall that we injected it into the DAO implementations). Spring Boot will also autoconfigure a DataSourceTransactionManager, so we don't need to do any other configuration.

_5. Next, take a quick look at the provided web layer (you don't need to modify anything):

a. The *com.oaktreeair.ffprogram.controllers.FlierController* autowires a reference to the FrequentFlierProgram facade layer, and responds to GET requests ending in */redeemaward*.

The *redeemAward()* method uses the facade to retrieve the balance before redemption, then redeems an award and retrieves the "after" balance. It stores the results of each operation as model variables for display by the redeemresult.jsp.

b. The redeemresult.jsp simply displays the results using JSP expressions.

_6. To test, use *gradlew bootRun* to start the application, then browse to *http://localhost:8080/redeemaward*. You should then use the H2 console to verify that the new award was inserted.

_7. Stop the application, then edit *application.resources* and add:

```
logging.level.org.springframework.jdbc.datasource=DEBUG
```

Then run the program again and browse to the same URL. Scroll back in the command prompt - you should see the DataSourceTransaction manager beginning and committing the transactions.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 15: Hello, World Web SOAP Service

In this lab, you will write the "Hello, world" of Web services to get you started in Web services development.

For this lab only, you will work outside of the development environment to demonstrate how JDK version 6 and later support JAX-WS Web services.

Objectives:

- To write a simple Web service
- To write a simple standalone Web service client

Part 1: Creating a Web Service

Steps:

_1. Ask your instructor for the following information:

Lab Installation directory: _____

If your computer is running Windows Vista, Windows 7 or later, the directory will be:

```
Lab Installation directory:   c:\Users\username\webclass
```

Note that you should substitute your actual user name for the "username" text above, e.g. "libtrain1".

_2. Stop any previous lab Spring Boot application - this lab will use port 8080, which conflicts with earlier labs.

_3. Open a command prompt window, then create the service:

a. Change the current directory in the command prompt to a subdirectory in the **{Lab Installation Directory}**.

If your computer has Windows Vista or later and the standard lab setup, enter:

```
c:
cd \Users\username\webclass\lab15\server
```

Be sure to substitute your actual user name.

_4. Use Notepad (or other editor) to create a new file named **HelloWorld.java**:

```
notepad service\HelloWorld.java
```

Complete the file by entering the following:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
package service;

import javax.jws.*;

@WebService
public class HelloWorld
{
    public String hello(String name)
    {
        return "Hello, " + name;
    }
}
```

Be sure to save the file.

Note: If you are using Notepad, when you save, set the "Save as type" to **All File** and enter **HelloWorld.java** for the file name. The HelloWorld.java file MUST be in the *service* folder.

_5. Use Notepad (or other editor) to create a file named **HelloPublisher.java**:

```
notepad service\HelloPublisher.java
```

Complete the file by entering the following:

```
package service;

import javax.xml.ws.*;

public class HelloPublisher
{
    public static void main(String[] args)
    {
       Endpoint.publish("http://localhost:8080/Hello",
           new HelloWorld());
       System.out.println("Service running...");
    }
}
```

Be sure to save the file.

_6. Compile the service code:

```
javac service\*.java
```

_7. Start the service:

```
java service.HelloPublisher
```

You should see a "Service running" message.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

_8. Test that your service is running correctly by examining the service description (WSDL). Open a Web browser and navigate to:

```
http://localhost:8080/Hello?WSDL
```

You should see the XML for the service description. See if you can make any sense of it.

_9. Open a new command prompt window, then change to a directory in which you will write a client.

If your computer has Windows Vista or later and the standard lab setup, enter:

```
c:
cd \Users\username\webclass\lab15\client
```

Be sure to substitute your actual user name.

_10. Generate a "stub" for your service from the WSDL:

```
wsimport http://localhost:8080/Hello?WSDL
```

_11. Examine the generated stub classes:

```
dir service
```

The **HelloWorld.class** file is a Java interface that represents the Web service. The **HelloWorldService.class** file is a Java class that acts as a "factory" for retrieving a stub object.

_12. Use Notepad (or other editor) to create a file named **HelloClient.java**:

```
notepad main\HelloClient.java
```

Complete HelloClient.java by entering the following:

```
package main;

import service.*;

public class HelloClient
{
   public static void main(String[] args)
   {
       HelloWorldService svc = new HelloWorldService();

       HelloWorld stub = svc.getHelloWorldPort();

       String reply = stub.hello("My name");
       System.out.println("Return from service: " + reply);
   }
}
```


Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Substitute your name for "My name".

_13. Compile your client:

```
javac main\*.java
```

_14. Run the client:

```
java main.HelloClient
```

You should see the reply from the service.

_15. You can clean up by closing all of the command windows. Be sure to close the window where the service is running so it won't conflict with Tomcat in later labs.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 16: Simple RESTful Service with JAX-RS

In this lab, you will create a simple RESTful service using JAX-RS.

Objectives:

- To demonstrate RESTfulness.

Part 1: Creating the Service

Steps:

_1. Switch to the Java EE Perspective.

_2. Create a new Dynamic Web project named **lab16**.

_3. Import all of the JARs from **{Lab Installation Directory}/lib/jersey** into the new project's **WebContent/WEB-INF/lib** folder.

_4. Create a new class named **HelloREST** in the *services* package.

_5. Complete the class so it can act as a source of RESTful content:

a. Write a JAX-RS annotation on the class that assigns a RESTful URL:

```
@Path("/hellorest")
```

b. Write a simple method that returns content:

```
public String hello(String name)
{
    return "Hello, " + name;
}
```

c. Annotate the method so that it responds to an HTTP *GET* and produces a text HTML result:

```
@GET
@Produces("text/html")
```

d. Annotate the *name* parameter so it's accessible to the service as an HTTP GET parameter:

```
@QueryParam("name")
```

e. Import types from the *javax.ws.rs* package.

_6. Create another class in the *services* package named **HelloApplication** that extends *javax.ws.rs.core.Application*.

Complete the class:

a. Annotate the class to provide a prefix for the RESTful URL:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
@ApplicationPath("/rest/")
```

b. Use the *Source - Override/Implement Methods* menu to override the *getClasses* method.

You can delete the generated call to the superclass.

c. Add code to the method so that it creates a Set containing all of your RESTful sources (only one in this case):

```
Set<Class<?>> s = new HashSet<Class<?>>();
s.add(HelloREST.class);
return s;
```

_7. Deploy the application by adding it to the Tomcat server, then restarting Tomcat.

_8. Test the RESTful service by opening a Web browser and entering this URL:

```
http://localhost:8080/lab16/rest/hellorest?name=xxxx
```

Substitute your name for the xxxx. You should see a simple Web page saying "Hello" to your name.

_9. Remove all projects from the Tomcat server and then stop it.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Optional Lab 17: RESTful CRUD Service using JAX-RS

In this lab, you will create a JAX-RS RESTful service that supports the some of the CRUD (Create, Retrieve, Update and Delete) operations using JAX-RS.

Objectives:

- To write a RESTful CRUD application using JAX-RS.

Part 1: Creating the Service

Steps:

_1. Create a new Dynamic Web project named **lab17**. On the last page of the wizard, be sure to put a checkmark in the box to create the Web deployment descriptor.

_2. Import all of the JARs from **{Lab Installation Directory}/lib/jersey** into the new project's **WebContent/WEB-INF/lib** folder.

_3. Next, create a Customer class that your service will retrieve and store:

a. In a new **com.goliath** package, create a class named **Customer**.

b. Decorate the class definition with the **XmlRootElement** annotation and import javax.xml.bind.annotation.XmlRootElement.

c. Add the following fields:

```
private int id;
private String name;
private String address;
private String city;
private String state;
private String zip;
```

d. Generate get/set methods for the fields.

e. Use *Source - Generate hashCode() and equals()* to generate those methods. Deselect all of the fields in the wizard except for the *id* field.

f. Use *Source - Generate toString()* to generate that method that displays all of the fields.

_4. Next, let's write a Web application context listener that will create an in-memory "database" of Customer objects and store them at application scope:

a. Right-click on the project and create a new class in the *com.goliath.listeners* package named **MyContextListener** that implements *ServletContextListener* from the *javax.servlet* package.

b. Annotate the class with **@WebListener** - this registers the listener.

c. In the *contextInitialized* method, write the following to create a Customer list and store it at application scope:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
List<Customer> customers = new ArrayList<Customer>();

Customer c = new Customer();
c.setId(12);
c.setAddress("123 Elm");
c.setCity("Madison");
c.setName("Bill Smith");
c.setState("WI");
c.setZip("53220");
customers.add(c);

c = new Customer();
c.setId(44);
c.setAddress("456 Oak");
c.setCity("Marion");
c.setName("Sue Jones");
c.setState("IA");
c.setZip("52302");
customers.add(c);

sce.getServletContext().setAttribute("customers", customers);
```

Part 2: Implementing the R in CRUD

In this part, you will implement retrieving Customer as JSON, given a customer ID.

Steps:

_1. First, create the Java class that will act as the service's endpoint:

a. In the **com.goliath.service** package, create a class named **CustomerService**.

b. Annotate the class to assign a URI to the service:

```
@Path("/customer/{id}")
```

Note that we use the "{id}" syntax to specify a placeholder in the URI for the customer's ID. In other words, URIs from the client will look like:

```
http://localhost:8080/lab17/rest/customer/xxxxx
```

Where 'xxxxx' is the customer ID.

Import the *javax.ws.rs.Path* type.

c. Define a field and annotate it so your service can access the ServletContext object that represents application scope:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
@Context
private ServletContext servletContext;
```

Import the *javax.servlet.ServletContext* and *javax.ws.rs.core.Context* types.

d. Start writing a method that will provide the "R" in CRUD, i.e. it will retrieve a customer in JSON form, given a customer ID:

```
@GET
@Produces("application/json")
public Response find(@PathParam("id") int id)
{
    Response ret = null;

    return ret;
}
```

Import *javax.ws.rs.core.Response* and *javax.ws.rs.Produces*.

Note how we annoate this method: @GET and @Produces configure the HTTP verb and content type and @PathParam maps the "{id}" URI template to the parameter on the method.

e. In the method, after the *String ret = null;* line but before the *return*, retrieve a reference to the customer list that your listener stored at application scope:

```
List<Customer> customers = (List<Customer>) servletContext
        .getAttribute("customers");
```

f. Then iterate the list, looking for a customer with the input ID:

```
Customer theCust = null;
for(Customer c : customers)
{
    if (c.getId() == id)
    {
        theCust = c;
        break;
    }
}
```

g. Next, write an *if-else* statement that checks if a there was a customer with the specified ID:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
if (theCust != null)
{

}
else
{

}
```

h. If theCust isn't null, use JAXB automatic object mapping to create an JSON response from the customer object:

```
ret = Response.ok(theCust).build();
```

i. In the *else* block (for theCust == null), send back HTTP status code "404 Not Found":

```
ret = Response.status(
        Response.Status.NOT_FOUND).build();
```

_2. Next, in the *com.goliath.service* package, create a class named **CustomerApplication** that extends *Application* from the *javax.ws.rs.core* package.

Complete the class:

a. Annotate the class to provide a prefix for the RESTful URL:

```
@ApplicationPath("/rest/")
```

b. Use the *Source - Override/Implement Methods* menu to override the *getClasses* method.

You can delete the generated call to the superclass.

c. Add code to the method so that it creates a Set containing all of your RESTful sources (only one in this case):

```
Set<Class<?>> s = new HashSet<Class<?>>();
s.add(CustomerService.class);
return s;
```

_3. Deploy the application by adding it to the server and then restarting the server.

_4. Test the RESTful service by opening a Web browser and entering this URL:

```
http://localhost:8080/lab17/rest/customer/44
```

You should see a simple Web page showing the JSON for customer #44.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

_5. Modify the URL so that the customer ID is 566 - you should get a 404 Not Found, but the browser may not report it. To see the actual request and response, configure Eclipse's TCP/IP monitor:

a. From the Eclipse menu, choose *Window - Preferences*.

In the "type filter text" box, start typing **TCP**, then click on TCP/IP Monitor.

b. Press the Add button, then enter:

```
Local monitoring port:   8081
Host name:               localhost
Port:                    8080
```

Leave other settings as is, but put a checkmark on the *Start monitor automatically* box. Press OK to close the preferences.

c. Back in your browser, modify the URL so the port number is 8081 instead of 8080 - this will send the request to the monitor, which will capture the request, send it to the service and then capture the result.

Back in Eclipse, look near the Console tab for the TCP/IP tab and maximize it - you should see the request/response, but they won't be very interesting.

To see the interesting HTTP headers, on the monitor tab's top-right, find the toolbar button that looks like a downward-pointing triangle (View Menu). Click that button and select *Show Header* - you should see the GET request and the HTTP/1.1 404 Not Found response.

Repeat testing with the customer ID of 44 and verify that the request/response make sense.

_6. **Optional**: Using the sample in the course notes as a guide, implement as many of the other CRUD methods as you have time for. To test PUT and DELETE, you will need to write a standalone client program, since browsers will not generate those HTTP methods.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 18: Spring MVC RESTful Service

In this lab, you will create a simple RESTful service using Spring MVC.

Objectives:

- To demonstrate RESTfulness with Spring MVC.

Part 1: Creating the Service

Steps:

_1. Open a command-prompt window and change to **{Lab Installation Directory/workspace/lab18**.

Then use the Spring Initializr to create a Spring Boot project with dependencies of *web*, *jdbc* and the *h2* database:

```
spring init -b 1.3.1.RELEASE -d=web,jdbc,h2 --build gradle -x
```

Use *gradlew* to generate an Eclipse project and then import it into Eclipse.

_2. In Eclipse, continue setting up the new project:

a. Rename the generated *com.example.demo* package to **com.oaktreeair.ffprogram**.

b. Rename the generated *DemoApplication* class to **FrequentFlierProgram**.

c. Create a new package named **com.oaktreeair.ffprogram.domain**, then copy and paste the *Segment* class from the *lab06JavaConfig* project into the new package.

_3. Next, create an SQL script that will create a database table for Segments and populate it with a couple of rows. In the *src/main/resources* folder, create a file named **schema.sql** with the following contents (you can copy and paste from the starters.txt file in **{Lab Installation Directory}/starters/spring/lab13}**):

```
CREATE TABLE Segment
(
   SegmentNumber   INTEGER NOT NULL IDENTITY,
   SegmentDate     DATE,
   FlightNumber    INTEGER,
   OrigCity        VARCHAR(10),
   Miles           INTEGER,
   CONSTRAINT  PK_Item  PRIMARY KEY(SegmentNumber)
);


INSERT INTO Segment VALUES (NULL, '2009-10-18', 333, 'SFO', 367);
INSERT INTO Segment VALUES (NULL, '2009-10-18', 745, 'LAX', 1900);
INSERT INTO Segment VALUES (NULL, '2009-10-22', 453, 'BOS', 1900);
INSERT INTO Segment VALUES (NULL, '2009-10-18', 112, 'LAX', 367);
```


Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

_4. Copy and paste from the *lab13* project, copying the *com.oaktreeair.ffprogram.dao* package. If you did not finish lab13, ask the instructor for help in obtaining the solution.

Note: You may need to update the imported classes since the Segment class is in a different package here than in lab13.

_5. Add a new method to the SegmentDao interface and the DAO implementation to find a segment given a segment number.

_6. Right-click on the project and create a new class in the *com.oaktreeair.ffprogram.controller* package named **SegmentController**.

Complete the new class:

a. Annotate the class so that it's a Spring MVC controller with a request-mapping URL of **/rest**.

b. Define a field and autowire a reference to the DAO.

c. Create a private, static inner class named **ResourceNotFoundException** that extends RuntimeException. Annotate the exception class with:

```
@ResponseStatus( value = HttpStatus.NOT_FOUND )
```

d. Write a service method that responds to a URL ending in **/segment/{segmentNumber}** for an HTTP GET request. The method should accept the segment number as a parameter and use the DAO to find the Segment and then return it.

If there's no Segment with the specified segment number, throw the ResourceNotFoundException. Note that the DAO will throw an EmptyResultDataAccessException in that case, so you can catch that exception in the controller and rethrow it as ResourceNotFoundException.

_7. Back in the command prompt, use *gradlew bootRun* to start the application.

_8. Open a web browser and enter this URL:

```
http://localhost:8080/rest/segment/1
```

You should see a JSON document for the first row in the SEGMENT table.

Try again with a URL for Segment #59 - you should get a Not Found response.

_9. In the *src/main/resources/static* folder, create an HTML file named **view-segment.html**. In it, write a React page that uses axios to retrieve Segment #1 and display it. Here are some hints and suggestions:

- You should be able to "borrow" from the lab05 projects.
- Define a React component named *ShowSegment* that accepts a Segment in its *props*.
- As you did in lab05, Part 2, first render some placeholder text, then use axios to GET the JSON for Segment #1. When the promise resolves, render your ShowSegment component, passing the returned Segment.
- Test by re-starting the application with *gradlew bootRun*, then browsing to *http://localhost:8080/view-segment.html*.

Part 2: Implementing CRUD and Writing a React/axios Ajax Client

In this part, you will finish the service so it provides the following:

- Support for retrieving all segments.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

- Support for HTTP POST to create a new segment.
- Support for HTTP PUT to update an existing segment.
- Support for HTTP DELETE to remove a segment

You will then write React pages that use axios Ajax to exercise the service.

Steps:

_1. Start by updating the controller so it responds to GET request to view all segments. Then write a React page named **view-allsegments.html** that displays the segment list in a React component that renders an HTML table. You can borrow inspiration from the *view-fliers.html* page you wrote back in lab05part2.

_2. Update the controller and DAO by writing methods that CREATE a new segment, UPDATE an existing segment and DELETE a segment.

Test with cURL and/or a browser.

_3. Write React pages, each of which exercises one of the service methods using the axios Ajax support. For the methods that accept parameter(s), we recommend that you write React controlled-component forms so the user can enter the parameter data. Look back in the Introduction to React chapter for an example.

Note: Axios provides get(), post(), put() and delete() methods. Here's an example of post() (update() and delete() are similar):

```
axios.post('/rest/people',
  {
    firstName: 'Fred',
    lastName: 'Flintstone'
  })
  .then(function (response)
  {
    console.log(response.headers.location);
  })
  .catch(function (error)
  {
    console.log(error.message);
    if(error.response)
    {
       console.log(error.response.status);
    }
  });
```


Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 19: Additional Spring Boot Topics

In this lab, you will continue working with Spring Boot.

Objectives:

- To customize Spring boot starter dependencies
- To customize Spring boot autoconfiguration
- To deploy to a WAR

Part 1: Working with Starter Dependencies

In this part, you will create and run a Spring Boot Web application that uses a database.

Steps:

_1. Open a command-prompt or Terminal and change to **{Lab Installation Directory}/lab19**.

_2. Use the following command to browse the starter dependencies available to the Spring Boot CLI:

```
spring init --list | more
```

Hint: Use the space bar to advance pages in the output.

Note especially the **h2** (H2 database), **jdbc** (Spring JDBC support), **web** (Spring MVC) and **thymeleaf** starters.

_3. Use the Spring Boot CLI to generate a starter project with these specifications:

```
Dependencies:          web, h2, jdbc
Build language:        Gradle
Options:               Expanded project
Spring Boot version:   1.3.1.RELEASE
```

_4. Use Windows WordPad to edit the *build.gradle* and note the starter dependencies. Especially note how there's no version numbers - Spring Boot will find versions that work well together.

_5. Now let's examine which versions of the dependencies (and THEIR dependencies) that Spring Boot will bring in. In your command prompt:

a.
```
gradlew dependencies > dependencies.txt
```

This will first build the project and then emit a large tree of the libraries and their versions into the text file (there will be no output). Open dependencies.txt into WordPad.

b. Find the section titled **runtime - Runtime classpath for source set 'main'**, and identity the versions of:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

org.springframework:spring-jdbc: _____

com.fasterxml.jackson.core:jackson-databind: _____

com.h2database:h2: _____

org.yaml:snakeyaml: _____

_6. Note how the starter for Web automatically brought in the Jackson library, which converts Java objects to/from JSON. In our application, however, we will not use JSON, Jackson isn't really necessary, so let's exclude it:

a. In build.gradle, in the *dependencies* configuration block, modify the starter-web dependency so it looks like:

```
compile('org.springframework.boot:spring-boot-starter-web') {
   exclude group: 'com.fasterxml.jackson.core'
}
```

b. Close the WordPad window displaying dependencies.txt.

c. Display the dependency tree again, this time just to the console, filtering using the Windows *find* command:

```
gradlew dependencies | find "jackson"
```

Note that the Jackson dependency is no longer present.

d. Repeat the process to exclude the *hibernate-validator* dependency. **Note:** The *group* name is **org.hibernate** (you can also exclude by *module* - the module name is **hibernate-validator**).

Display a filtered dependency tree to verify that "hibernate" is no longer present.

_7. Now let's imagine that our QA team has identified a critical bug for our application in the H2 database, and recommends that we use a specific version. Let's modify the build to use that version:

a. Since we have an explicit dependency on H2, we can directly specify the version number. Modify the H2 dependency so it looks like:

```
runtime('com.h2database:h2:1.0.60')
```

b. Display the dependency tree again, filtering for "h2" - it should show the required H2 version.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

_8. Next, lets specify a specific version of a *transitive* dependency. YAML provides an alternate syntax for specifying application properties, and the *jdbc* starter brings it in so we can configure database connections using YAML syntax via the snakeyaml library.

Our scenario is that we want to use a different, older version. To accomplish that in Gradle, we need to first exclude the suggested (newer) version, then explicitly specify the version we want:

a. Modify the *starter-jdbc* dependency so it looks like:

```
compile('org.springframework.boot:spring-boot-starter-jdbc') {
  exclude module: 'snakeyaml'
}
```

b. Add the following:

```
compile('org.yaml:snakeyaml:1.8')
```

c. Display the dependency tree - it should have the 1.8 version of snakeyaml after an arrow '->' listing the suggested version.

_9. Now lets build the Java domain classes for the application, which will display a list of customers.

a. Generate an Eclipse project and bring the existing project into Eclipse.

b. Rename the generated *com.example.demo* package to **com.goliath**.

Then rename the generated *DemoApplication* class to **GoliathCustomers**.

c. In a new *com.goliath.domain* package, create a class named **Customer** with the following fields.

```
private int customerID;
private String name;
private String email;
private String phone;
```

Generate get/set methods and a toString() method.

_10. Next, create the data layer for the application:

a. In a new *com.goliath.dao* package, create a new interface named **CustomerDao**.

Add the following method:

```
List<Customer> allCustomers();
```

b. Create a class named **CustomerDaoImpl** that implements the interface.

c. Annotate the class with **@Repository**.

d. As you have done in earlier labs, write an inner class that defines a RowMapper that maps JDBC rows to Customer objects.

The database you will configure shortly will have a table named **Customer** with the following schema:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
CREATE TABLE customer(customerID INT IDENTITY PRIMARY KEY,
  name VARCHAR(100), phone VARCHAR(100), email VARCHAR(100));
```

e. After the inner class, define fields in the DAO class:

```
private JdbcTemplate template;
private CustomerRowMapper mapper;
```

f. After the fields, inject the data source:

```
@Autowired
public void setDataSource(DataSource ds)
{
    template = new JdbcTemplate(ds);
    mapper = new CustomerRowMapper();
}
```

g. Complete the allCustomers() method so that it uses the JdbcTemplate to query and return a List of all Customers, using the row-mapper you coded earlier.

_11. Next, create the web layer:

a. In a new *com.goliath.controllers* package, create a class named **CustomerController**. Annotate the class so it's a Spring MVC controller responding to requests whose URL ends in **/customers**.

b. Inject a DAO reference into a field.

c. Write a method that sets up the customer list for display. The method should respond to a GET request, retrieve the Customer list from the DAO, then store it in the model with the name **customers**, then return the String **customers**.

In the next step, you'll write a JSP named *customers.jsp* that will access the *customer* list from the model and use the JSTL to iterate and display.

_12. Next, create the Web user interface:

a. Right-click on the project and create a new folder named **src/main/webapp/WEB-INF/jsp**.

b. In the new folder, create a JSP named **customers.jsp** with the following content:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
<%@ taglib prefix="c"
     uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Customer List</title>
    <link rel="stylesheet" href="/style.css"></link>
  </head>
  <body>
    <h1>Customer List</h1>
    <table>
      <tr>
        <th>Customer ID</th>
        <th>Name</th>
        <th>Phone</th>
        <th>Email</th>
      </tr>
      <c:forEach items="${customers}" var="cust">
        <tr>
          <td>${cust.customerID}</td>
          <td>${cust.name}</td>
          <td>${cust.phone}</td>
          <td>${cust.email}</td>
        </tr>
      </c:forEach>
    </table>
  </body>
</html>
```

c. In the *src/main/resources/static* folder, create a file named **style.css** with the following very simple CSS styling:

```
td, th { border: 1px solid; }
```

_13. Edit the *src/main/resources/application.properties* file and add:

```
spring.mvc.view.prefix: /WEB-INF/jsp/
spring.mvc.view.suffix: .jsp
```

This configures Spring MVC so it can find your JSPs.

Note: Please be sure there are no extra spaces at the end of each of those lines.

_14. Edit the *build.gradle* file and modify it:

a. In the *apply-plugin* list, add:

```
apply plugin: 'war'
```


Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

b. In the *dependencies* block, add:

```
providedRuntime("org.apache.tomcat.embed:tomcat-embed-jasper")
providedRuntime("javax.servlet:jstl")
```

_15. Next, create an SQL script that will create the Customer table and populate it with a couple of rows. In the *src/main/resources* folder, create a file named **schema.sql** with the following contents:

```
CREATE TABLE customer(customerID INT IDENTITY PRIMARY KEY,
  name VARCHAR(100), phone VARCHAR(100), email VARCHAR(100));

INSERT INTO customer (name, phone, email) VALUES
  ('Sam Smith', '555-1212', 'sam@sam.com');
INSERT INTO customer (name, phone, email) VALUES
  ('Sue Jones', '555-1313', 'sue@sue.com');
```

_16. Test the application:

a. In the same fashion as in earlier labs, in your command prompt, use *gradlew bootRun* to start an embedded Tomcat instance and your application. Leave the command prompt running.

b. Open a browser and enter this URL:

```
http://localhost:8080/customers
```

You should see your application's table of customer info with two rows of data.

Part 2: Working with Autoconfiguration and Deployment

In this part, you will rewrite the Customer list application so it provides the customer list as a RESTful resource.

Steps:

_1. Open a command-prompt and change to **{Lab Installation Directory}/lab19p2**.

_2. Use the Spring Boot CLI to generate a starter project with these specification:

```
Dependencies:         web, h2, jdbc
Build language:       Gradle
Options:              Expanded project
Spring Boot version:  1.3.1.RELEASE
```

_3. Edit the *build.gradle* and modify it:

a. In the *apply-plugin* list, add:

```
apply plugin: 'war'
```

b. In the *dependencies* block, add:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
providedRuntime("org.apache.tomcat.embed:tomcat-embed-jasper")
```

_4. Generate an Eclipse project and import it into Eclipse.

Then rename the generated *com.example.demo* package to **com.goliath** and rename the generated *DemoApplication* class to **GoliathCustomers**.

_5. Copy the following from the *lab19* project to equivalent locations in the new project:

- The *com.goliath.domain* package
- The *com.goliath.dao* package
- The *src/main/resources/schema.sql* file
- The *src/main/static/resources/static/style.css* file

_6. Next, create a controller that provides the customer list as a RESTful resource:

a. In a new *com.goliath.controllers* package, create a class named **CustomerController**.

b. Annotate the class so that it's a controller.

c. In the same fashion as in the last part of lab, autowire a reference to the DAO.

d. Write a method that returns all customers as a RESTful resource with GET URL of */customer*, fetching the Customer list from the DAO.

_7. As a quick test, in your command prompt, use *gradlew bootRun* to start the app, then open a browser and navigate to:

```
http://localhost:8080/customer
```

You should see the customer list in JSON format.

Stop the server.

_8. Next, in the *src/main/resources/static* folder, create a file named **view-customers.html** that defines a React component to display the customer list in an HTML table. Use *axios* to GET the JSON from the server.

_9. Use *gradlew bootRun* to run the app, then browse to:

```
http://localhost:8080/view-customers.html
```

You should see the customer list.

Part 3: Using an External Database

So far, your applications in this lab have used the H2 database in its in-memory mode, which is the default for Spring Boot. This mode is very fast for development, but doesn't actually save data persistently (Spring Boot ran the schema.sql script automatically each time you booted the app).

In this part, you will convert the customer-list application to use an external database.

Steps:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

_1. You must configure the database for your program. We will use the free, open-source HypersonicSQL database to hold the customer information. Follow these steps to configure the database:

a. Open a command-prompt window (choose Start - Run and type **cmd**), then change to the **{Lab Installation Directory}/lib** directory. If your computer has the standard lab setup:

```
c:
cd \Users\username\webclass\lib\hsql
```

b. Start the HypersonicSQL server running:

```
java -cp hsqldb-2.3.3.jar org.hsqldb.Server -database mydb
```

The server will start and display a sign-on message. If prompted by Windows, allow access for the server.

Note: Be sure to leave this command-prompt running until you are finished with the lab.

c. Open another command-prompt window and change to same directory as you did in the last step.

d. Start the HypersonicSQL administrator's GUI:

```
java -cp hsqldb-2.3.3.jar org.hsqldb.util.DatabaseManager
```

e. In the GUI, change the *Type* to **HSQL Database Engine Server**, then press OK. **Warning:** Be sure to choose the Server option!

f. In the GUI, in the command area between the Clear and Execute buttons, create and populate the database table, by entering the following (copy and paste from the 'starters' file):

```
CREATE TABLE customer(customerID INT IDENTITY PRIMARY KEY,
  name VARCHAR(100), phone VARCHAR(100), email VARCHAR(100));
```

Then press the Execute button.

Press the Clear button, then execute:

```
INSERT INTO customer (name, phone, email) VALUES
  ('Sam Smith', '555-1212', 'sam@sam.com');
INSERT INTO customer (name, phone, email) VALUES
  ('Sue Jones', '555-1313', 'sue@sue.com');
INSERT INTO customer (name, phone, email) VALUES
  ('Harry Wolfe', '555-1414', 'harry@wolfe.com');
```

To test the new table, press the CLEAR button, then enter:

```
SELECT * from customer
```

Press the Execute button -- you should see the customer info.

You can close the GUI before continuing, but be sure to leave the server running.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

_2. Use Windows Explorer to copy the *lab19p2* directory into the *workspace* folder with a new name of **lab19p3**. Then import the *lab19p3* project into Eclipse.

_3. In Eclipse, edit *build.gradle* and remove the H2 dependency line. Then add a dependency to the HSQLDB database:

```
runtime('org.hsqldb:hsqldb')
```

_4. Edit *src/main/resources/application.properties* and add the following lines to override the database autoconfiguration so that your application use the HSQLDB database instead of the in-memory H2 database:

```
spring.datasource.url=jdbc:hsqldb:hsql://localhost/
spring.datasource.username=SA
spring.datasource.password=
spring.datasource.driver-class-name=org.hsqldb.jdbcDriver
```

Note: Please ensure that there are no extra spaces at the end of any of these lines.

_5. Delete the *src/main/resources/schema.sql* file - we don't need it since we created and populated the table manually.

_6. Open a command-prompt or Terminal and change to **{Lab Installation Directory}/lab19p3**.

_7. Use *gradlew clean bootRun* to run the app, then browse to:

```
http://localhost:8080/customers.html
```

You should see the customer list, now getting its data from the external HSQLDB database.

Part 4: Deploying as a WAR

So far, you have tested your applications using Spring Boot's embedded Tomcat. In this part, you will create a Web archive (WAR) file and deploy it to a standalone Tomcat instance.

Steps:

_1. Use Windows Explorer to copy the *lab19p3* directory into the *workspace* folder with a new name of **lab19p4**. Then import the *lab19p4* project into Eclipse.

_2. Edit *view-customers.html*:

a. Since you will deploy as a WAR, the Customer JSON will be at a different URL. Modify your *axios* request to fetch from **http://localhost:8080/lab19p4/customer**.

b. If you used the *style.css* styleseet, ensure that there's no leading slash in the *href* that links to it.

_3. Since we will run the application as a WAR, not as an executable JAR, we need to ensure that the Spring MVC *dispatcher servlet* is registered with Tomcat. One way to do that is to create a subclass of SpringBootServletInitializer (another would be to create a web.xml file and register the servlet there):

a. In the *com.goliath* package, create a class named **MyServletInitializer** that's a subclass of *org.springframework.boot.context.web.SpringBootServletInitializer*.

b. Use Eclipse's *Source - Override/Implement Methods* menu to override the *configure()* method.

c. Modify the configure() method so it looks like:

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
@Override
protected SpringApplicationBuilder configure(
        SpringApplicationBuilder builder)
{
    return builder.sources(GoliathCustomers.class);
}
```

_4. Open a command-prompt and change to **{Lab Installation Directory}/lab19p4**.

_5. Use the *gradlew clean build* command to build the WAR, the

Look in the *build/libs* folder - you should see the WAR file.

_6. Next, deploy the WAR to the standalone Tomcat server you used in earlier labs:

a. Verify that Tomcat is installed in the *c:\Users\username\java\apache-tomcat-8.0.xx* directory. Note that the xx is a placeholder for the Tomcat version actually installed. We will refer to this as the **Tomcat Installation Directory**.

b. Use Windows Explorer to copy *lab19p4-0.0.1-SNAPSHOT.war* to **{Tomcat Installation Directory}/webapps**. This deploys the application.

Change the name of the copied file to **lab19p4.war**. Tomcat uses the WAR name as the application's *context root*, which is part of the app's URL.

c. Use Windows Explorer to copy the **{Lab Installation Directory}/lib/hsql/hsqldb-2.3.3.jar** file to the **{Tomcat Installation Directory}/lib** folder. This makes the HSQLDB driver available to Tomcat.

_7. Ensure that the HSQLDB server is still running and if not, start it as described in the last part of lab.

_8. Now you can start Tomcat:

a. Open a command prompt and change to **{Tomcat Installation Directory}** directory.

b. Start Tomcat:

```
bin\catalina.bat run
```

Tomcat should start and in its console, show your app's deployment.

_9. Test your application by opening a browser and navigating to:

```
http://localhost:8080/lab19p4/view-customers.html
```

You should see the customer list.

Be sure to kill the external Tomcat instance and the HSQL server and GUI before continuing.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Case Study Lab

In this lab, you will write a complete web application that uses most of the techniques you learned in the class. We do not expect you to finish this lab in the time allotted, but you should be able to architect the system and start working on it.

Here's the situation. You work in Information Technology for *Bike Trails Forever*, a large, non-profit organization that accepts donations from individuals. In the past, BTF has used a PC-based system to keep a record of donations, but in this day and age, the officers are demanding a new, Web-based solution. You have been given the job of creating such a system using Spring and a database.

Here are the specifications:

- For each donation, you must track the date and amount, as well as the donor's name and address information
- A single individual might make several donations over time -- your program needs to track each one separately
- Your database analyst has recommended using two database tables related by a *foreign key*. One table will hold the name information and the other will store the donation information. To relate the tables, the donation table should store a foreign key to the name table
- Your goal is to design a single-page Web application that uses Spring and RESTful services on the server side and React and axios on the client. The page should list all of the donors in a table and all of the donations for a given donor in a separate table. If you have time, you can also provide an input form to enter a new donation and associate it with a donor.

Note to the instructor: it's useful to do the design part of this lab (Step One below) as a group activity. Divide the class into small groups (three or four students each) and have each group come up with a design. Then optionally have each group present their design in front of the class.

Objectives:

- To design and implement a web application from scratch

Steps:

_1. Before you start banging out code, you should spend some up-front time on the design and architecture of your system. Here's what we recommend (not necessarily in this order):

   - Sketch out your HTML screens and storyboard the sequence of how they will work.
   - Sketch out what columns the database tables should have.
   - Do some object-oriented analysis and design to come up with the domain class(es) that you will need for the model. What will their attributes and methods look like?
   - Think about what Spring beans and DAOs that your program will need. What RESTful services will you need?
   - Think about how transactions should work.
   - Write some pseudo-code (or better yet, a UML interaction diagram) for each use of the system that you will implement. Think about what actions you will need to perform on the database for each use (e.g. insert versus update).

_2. Since the goal of this lab is to let you practice the techniques you learned in class, this lab guide will not give you step-by-step instructions as in earlier labs.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

_3. Use Spring Boot to generate a starter project.

You will need to create the database. For purposes of this lab, it's OK to use H2 in its in-memory mode.

You will also need to define a database table to hold the data. Use the SQL scripts from previous labs as inspiration.

_4. Good luck!

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.