Core Spring Framework Workshop

Lab Exercises

Eclipse Mars/Tomcat 8/Spring4.2.6/Gradle Version

Version 1.0.2 June 14, 2016

You can contact Descriptor Systems at:

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

Core Spring Framework Workshop Labs

Lab 1: 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.

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. Ask your instructor for the following information:

Tomcat Installation directory: _____

Lab Installation directory: _____

Eclipse Installation directory: _____

If your computer is running Windows Vista or later, the directories will be something like:

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

Substitute your actual username for the placeholder "username".

_2. Use Windows Explorer to navigate to the **{Eclipse Installation Directory}**. Launch Eclipse by double-clicking on the **eclipse.exe** file.

When Eclipse prompts you, enter **{Lab Installation Directory}/workspace** for the *Workspace location*.

On Windows Vista or later:

```
c:\Users\username\springclass\workspace
```

Once Eclipse starts, you can close any Welcome window that appears.

_3. Eclipse supports the notion of *perspectives*, which are sets of window panes that let you work in a particular role, e.g. Java coder, Web developer, server administrator and so forth. For this lab, you will work in the *Java* perspective, which contains window panes that let you create and edit Java pplications.

Open the *Java perspective* by choosing **Window - Open Perspective - Java**.

_4. Next you can configure and customize Eclipse. Choose from the menu **Window - Preferences** -- this opens the Preferences property sheet where you can customize nearly all aspects of Eclipse. You can play around with the settings as you wish.

For example, you can change the font used by code editors by selecting in the left-side tree view: **General - Appearance - Colors and Fonts** and then selecting **Basic - Text Font** and pressing the *Change...* button.

_5. 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\springclass\workspace\lab01
```

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
```

_6. 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 *lab01* 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.

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. 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.

_7. 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/lab01/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.

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

_8. 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.

_9. 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().

_10. 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:

```
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);
}
```

_11. 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".

_12. Next, edit ContactInfo.java and repeat the above step to initialize the ContactInfo's fields to values of your choice.

_13. 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.

_14. 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.

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.

_15. 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;
}
```

_16. 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"/>
```

_17. 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.

_18. 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.

_19. Edit FrequentFlierProgram.java and test your bonus calculation:

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.

_20. 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**.

_21. 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.

_22. Now let's experiment with more verbose logging. Edit the *log4j.properties* file and change the first line so it looks like:

```
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 lab01 project and choose *Copy*, then press Ctrl+V. Name the new project **lab01JavaConfig**.

_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:

```
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 lab01 project and choose *Copy*, then press Ctrl+V. Name the new project **lab01XML**.

_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.

Lab 2: 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. Ask your instructor for the following information:

Tomcat Installation directory: _____

Lab Installation directory: _____

Eclipse Installation directory: _____

If your computer is running Windows Vista or later, the directories will be something like:

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

Substitute your actual username for the placeholder "username".

_2. 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\springclass\workspace\lab02
```

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
```

_3. 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 *lab02* 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.

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. 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.

g. Copy and paste the *spring.xml* file from the lab01 project into the new project's *src/main/java* folder.

Then update the *component-scan* so its base-package is **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. 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. 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);
```

_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 void applyForLoan(Borrower b)
{
    boolean rating = loanService.processLoan(b);

    System.out.println("Loan status is " +
        b.getTheLoan().getStatus());

    if (rating)
    {
        System.out.println("Here's a free toaster!");
    }
    else
    {
        System.out.println("Sorry, no loan today");
    }
}
```

_6. Create a new class named **LoanApplication** in the *com.bigbank.test* package with a *main* method. Complete the *main* method:

   a. Create a Spring container, referencing the spring.xml configuration file.

   b. Retrieve a reference to the Borrower bean from the container.

   c. Retrieve a reference to the LoanManager bean from the container.

   d. Call the LoanManager's applyForLoan() method.

_7. Run LoanApplication as a Java application. You should see the loan-application outcome.

Lab 3: 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 *lab02* and choose *Copy*. Then right-click on a blank area in the Package Explorer and choose *Paste*, naming the copied project **lab03**.

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

   a. In the *lab03* 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. Open the spring.xml file and delete the *context:component-scan* element.

_3. Try running the LoanApplication class as a Java application - 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.

      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. Modify the LoanApplication main method so it loads configuration from the LoanManagerConfig class rather than the XML file.

_5. Run LoanApplication as Java program. 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 Part 2: 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.

Then add code to LoanApplication's main() method to retrieve the loan02 bean and display its properties.

Lab 4: Spring Beans

In this lab, you will work with various aspects of Spring beans.

Objectives:

- To work with the bean lifecycle
- To write a BeanPostProcessor
- To configure beans that use a factory class

Part 1: Bean Lifecycle

In this part, you will experiment with the Spring bean lifecycle in the Segment bean. You will write an initialization method that creates a random number and stores it in a temporary file. You will also write a destruction method that deletes the temporary file.

Steps:

_1. Follow this procedure to make a copy of your lab01 project:

   a. Choose *File - Close All* to close all open editors.

   b. In the Project Explorer, right-click on the *lab01* project and choose *Copy*.

   c. Right-click the blank, white area of the Project Explorer and choose *Paste*.

   In the *Copy Project* dialog, enter **lab04** and press OK.

_2. Open the Segment.java class into the editor. Write a method that will acts as the *init* method:

```
@PostConstruct
public void init()
{
    System.out.println("In init() method");
    try
    {
        Random rand = new Random();
        long randID = rand.nextLong();
        DataOutputStream dos = new DataOutputStream(
           new FileOutputStream(
                "temp.dat"));
        dos.writeLong(randID);
        dos.close();
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
}
```

This method calculates a random number for a segment ID and writes it to a "temporary" file. Not how we annotate it so that it acts as an "init" method.

_3. Modify the *getSegmentNumber* method so that it reads the random ID from the temporary file:

```
public Long getSegmentNumber()
{
    Long segNum = null;
    try
    {
        DataInputStream dis = new DataInputStream
          (new FileInputStream(
                "temp.dat"));
        long val = dis.readLong();
        segNum = val;
        dis.close();
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
    return segNum;
}
```

_4. Now update the application to use the Segment bean and run it:

a. In FrequentFlierProgram.java, in *main*, retrieve the *seg01* bean from the application context, then display its *segmentNumber* property (the random number) to the console.

b. Run the FrequentFlierProgram.java class as a Java application - you should see the random segment number.

c. In the Project Explorer, right-click on the *springlab04* project and choose *Refresh* - you should see the "temporary" file. You will clean that up in the next step.

_5. Delete the temporary file as part of the bean's destruction:

a. In Segment.java, write a new method to delete the file:

```
@PreDestroy
public void destroy()
{
    System.out.println("In destroy() method");
    File temp = new File("temp.dat");
    temp.delete();
}
```

b. In FrequentFlierProgram.java, ensure that at the end of *main*, you have code to close the application context.

c. Run FrequentFlierProgram.java, then refresh the project in the Project Explorer. If your destroy-method worked, the temporary file should be gone.

Part 2: Writing a Bean Post Processor

In this part, you will write a BeanPostProcessor that injects a temporary file reference into the Segment bean, or any other bean that needs to use a temporary file.

Steps:

_1. In the *com.oaktreeair.ffprogram* package, create a new interface named **TempFileAware**. Add the following method to the interface:

```
void setTempFile(File temp);
```

_2. Open Segment.java into the editor and make the following changes:

a. Modify the class definition so the class implements the TempFileAware interface.

b. Define a field:

```
private File tempFile;
```

c. Write the *setTempFile* method from the interface. It should store the passed File reference in the field you just defined.

d. Modify the *init*, *getSegmentNumber* and *destroy* methods to use the temp file from the field rather than opening a new one. For example, the *init* method might look like:

```
public void init()
{
    try
    {
        Random rand = new Random();
        long randID = rand.nextLong();
        DataOutputStream dos = new DataOutputStream(
           new FileOutputStream(
                tempFile));
        dos.writeLong(randID);
        dos.close();
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
}
```

_3. Create a new class in the *com.oaktreeair.ffprogram.util* package named **TempFileBeanPostProcessor**. This class should implement the *BeanPostProcessor* interface.

Eclipse created empty methods for the interface methods, but perhaps did not give the arguments useful names. We recommend that you change the argument names to **bean** and **name**.

Since we will not substitute a different bean, modify both method's return statements to return the original input bean reference.

_4. Annotate the the class with @Component.

_5. Complete the new class's postProcessBeforeInitialization method (be sure to modify the "before" method instead of the "after"):

   a. First check to see if the input bean is an instance of the *TempFileAware* interface.

   b. If so, cast the reference to **TempFileAware**.

   c. Create an instance of a temporary file:

```
File tempFile = File.createTempFile("data", null,
        new File(System.getProperty("user.dir")));
```

   d. Call the *setTempFile* method on the bean, passing the *tempFile* reference. Then output an informative diagnostic message to the console indicating that you injected a temp file, using the bean name.

   You will need to write a *try-catch* block and import the appropriate types.

_6. Run FrequentFlierProgram.java. It should work as before.

Optional Part 3: Using a Factory Class with XML Configuration

In this part, you will inject a standard Java type, a *DocumentBuilder* into your Segment bean using XML "setter" injection.

The DocumentBuilder lets you parse XML content using the Document Object Model (DOM).

The standard Java (non-Spring) code to create a DOM parser and parse a file looks like:

```
DocumentBuilderFactory dbf =
   DocumentBuilderFactory.newInstance();
DocumentBuilder db = dbf.newDocumentBuilder();
Document doc = db.parse ( "flights.xml" );
```

The thing that makes this interesting for Spring is that the DocumentBuilder requires use of a factory class, DocumentBuilderFactory, to create instances. So your job is to configure this factory class and then inject a DocumentBuilder instance into a Segment.

The Segment bean will use the DocumentBuilder to parse an XML file and retrieve data from the XML.

Steps:

_1. Copy the *lab01XML* project into a new project named **lab04XML**.

_2. To get started, do a File System import into the *lab04XML* project's root directory, importing from **{Lab Installation Directory}/starters/lab04**, importing the **flights.xml** file:

   a. Use Windows Explorer or My Computer to navigate to **{Lab Installation Directory}/starters/lab02.**

   b. In Windows Explorer, right-click on *flights.xml* and and choose *Copy* to copy to the clipboard.

   c. Back in Eclipse, right-click on the *lab04XML* project's root folder and choose *Paste*.

   Open this file and note that it defines a couple of flights and information about them.

_3. Modify the Segment.java bean and define a field:

```
private DocumentBuilder parser;
```

Import the appropriate type and generate get/set methods for this field so the Spring container can inject into it.

_4. Next, configure the DocumentBuilderFactory in the Spring configuration file:

a. Open *spring.xml* into the editor and define a bean for the DocumentBuilderFactory:

```
<bean class="javax.xml.parsers.DocumentBuilderFactory"
    id="domFactory" factory-method="newInstance">
</bean>
```

b. Next, configure a DocumentBuilder instance, using the factory:

```
<bean id="theParser" factory-bean="domFactory"
    factory-method="newDocumentBuilder" />
```

c. Next, update the "seg01" bean's configuration to use setter injection to inject the parser reference. Also, inject values for the *flightNumber* and *originatingCity* properties that match values in flights.xml. In addition, configure an *init* method named **init**.

The completed configuration should look something like:

```
<bean id="seg01" init-method="init"
   class="com.oaktreeair.ffprogram.Segment">
    <property name="parser" ref="theParser" />
    <property name="flightNumber" value="333"/>
    <property name="originatingCity" value="SFO"/>
</bean>
```

_5. Next, write the Segment.java *init* method to use the parser to parse the flights.xml file and get the miles for a segment:

a. Write the skeleton of a method named **init** that accepts no parameters and returns void.

b. In the *init* method, write a try-catch block that catches generic **Exception**.

c. In the *try* block, write code to use the injected DOM parser to parse the flights.xml file:

```
Document doc = parser.parse ( "flights.xml" );
```

d. Immediately following the above line, create a String with an *XPath* expression to navigate to the correct flight:

```
StringBuilder sb = new StringBuilder();
sb.append("/flights/flight[@flightNumber='");
sb.append(flightNumber);
sb.append("']/segments/segment[@startCity='");
sb.append(originatingCity);
sb.append("']/miles");
String xpathStr = sb.toString();
```

Note: Be very careful to type this in correctly, especially making sure that you get the single and double quotes correct.

e. Compile the String into an *XPath* string:

```
XPath xpath = XPathFactory.newInstance().newXPath();
XPathExpression expr = xpath.compile(xpathStr);
```

f. Execute the XPath and retrieve the result, which is the miles for the specified flight and originating city:

```
Object result = expr.evaluate(doc, XPathConstants.NODESET);
NodeList nodes = (NodeList) result;
Node node = nodes.item(0);
miles = Integer.parseInt(node.getTextContent());
```

Note that we store the miles in the Segment's field.

Import types from the *org.w3c.dom* package as necessary, except *XPathExpression*, which is from the *javax.xml.xpath* package.

_6. Open FrequentFlierProgram.java into the editor and at the end of the *main* method, write code to display the *miles* property on the *seg01* bean.

Run the program - you should see the miles: 445.

Lab 5: XML Dependency Injection

In this lab, you will work with the basics of Spring XML dependency injection.

Objectives:

- To work with XML dependency injection

Part 1: XML Dependency Injection

In this part, you will work with constructor injection and setter injection using XML.

Steps:

_1. This lab depends on successfully completing the basic parts of lab01XML. If you did not finish the basic part of thant lab, you should either finish it or ask the instructor to help you get the last lab's solution.

_2. Follow this procedure to make a copy of your lab01XML project:

   a. Choose *File - Close All* to close all open editors.

   b. In the Project Explorer, right-click on the *lab01XML* project and choose *Copy*.

   c. Right-click the blank, white area of the Project Explorer and choose *Paste*.

   In the *Copy Project* dialog, enter **lab05XML** and press OK.

_3. In the same fashion as before, create a new class in the *com.oaktreeair.ffprogram* package named **AddressInfo**.

Complete the class so it has the following properties, complete with get/set methods:

```
private String street;
private String city;
private String state;
private String zip;
private String country;
```

Then write a constructor in the new class that accepts parameters corresponding to each of the fields.

_4. In the *spring.xml* file, use *constructor injection* to define an AddressInfo bean with values of your choice. Assign the new bean an ID of **addrInfo01**.

_5. Modify the *Flier* class to define a property of type AddressInfo named **homeAddress**. Be sure to define a get/set method pair for the new property.

_6. In the *spring.xml* file, use *setter injection* to inject the *addrInfo01* bean reference into the new property for the *flier01* bean.

_7. In the FrequentFlierProgram.java file, add code to the *main* method to retrieve the home address from the flier and display its properties to the console.

_8. Run the FrequentFlierProgram.java class as a Java Application. You should see the home address.

Part 2: The @Required Annotation

In this part, you will work with required properties.

Steps:

_1. The ContactInfo bean you created in an earlier lab has four properties: emailAddress, homePhone, mobilePhone and smsNumber. Let's make the first two required properties:

a. Open ContactInfo.java into the editor. Find the *setEmailAddress* method, and immediately above it, annotate it as @Required:

```
@Required
public void setEmailAddress(String emailAddress)
{
    this.emailAddress = emailAddress;
}
```

Use *Source - Organize Imports* so that Eclipse imports org.springframework.beans.factory.annotation.Required.

b. Repeat for the *setHomePhone* method.

c. Edit *spring.xml* (where the contact01 bean is configured), and comment out the setter-injection lines for the emailAddress and homePhone properties.

d. Above any bean definition, add the following to enable annotation processing:

```
<bean class=
"org.springframework.beans.
factory.annotation.RequiredAnnotationBeanPostProcessor"/>
```

Note: The second line in the code above (the fully-qualified class name) is split so it fits on the printed page, but you must enter it unbroken.

e. Try running FrequentFlierProgram.java again - You should get a BeanCreationException. Uncomment the missing properties in spring.xml before continuing.

Part 3: Experimenting

In this part, you can experiment with your application. The following steps list things you can try.

Steps:

_1. Try changing the "flier01" bean's ID configuration to **"abc"** instead of a number. Run the main program and see what happens. Be sure to change it back.

_2. In the ContactInfo class, write a constructor that accepts values corresponding to all of the fields. Then, without changing any configuration, run the main program. What happens, and why? Fix the problem either by deleting the new constructor, writing a zero-argument constructor or by updating the configuration.

Lab 6: Spring and the Web

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

Objectives:

- To use basic servlets and JSPs as a Web front end
- To use SpringMVC

Part 1: Spring and the Web

In this part, you will create a servlet and JSP-based Web application that will act as the user interface for a Spring application. This part does not use SpringMVC, but does use a Spring *context listener* that makes the Spring application context available to the program.

Steps:

_1. Eclipse supports the notion of *perspectives*, which are sets of window panes that let you work in a particular role, e.g. Java coder, Web developer, server administrator and so forth. For this lab, you will work in the *Java EE* perspective, which contains window panes that let you create and edit JEE Applications.

Open the *JEE perspective* by choosing **Window - Open Perspective - Other - Java EE (default)**.

_2. Ask your instructor for the following information:

Tomcat Installation directory: _____

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

```
Tomcat Installation directory:
                              c:\Users\username\java\apache-tomcat-8.0.27
```

Substitute your actual username for the placeholder "username".

_3. Next, you will configure the connection between Eclipse and the Tomcat Web container:

a. Find the *Servers* tab at the bottom of the Eclipse window and click on it.

b. Right-click in the blank area on the Servers pane and choose *New - Server* to start the wizard.

c. On the first wizard page, expand the *Apache* category and select the *Tomcat v8.0 Server* entry, then press Next.

d. On the next wizard page, for the *Tomcat installation directory*, click the Browse button and navigate to your computer's **{Tomcat Installation Directory}**.

Press Next, followed by Finish. You should see the new server in the Servers pane.

_4. This lab depends on successfully completing the basic parts of *lab01*. If you did not finish the basic part of lab01, you should either finish it or ask the instructor to help you get the that lab's solution.

_5. 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\springclass\workspace\lab06Part1
```

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 dependencies on Spring 4.2.6 as well as spring-mvc, the servlet API, and the JSP Standard Tag Library (JSTL).

Also note that it specifies Gradle plugins to build an Eclipse WTP (Web Tools Project) project and build a Web archive (WAR).

d. Generate an Eclipse project and download the dependencies:

```
gradle eclipse
```

_6. 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 *lab06Part1* 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.

e. In the Project Explorer, expand the project and note that it contains the build.gradle script.

f. Right-click on the project and choose *Properties*, then click *Project Facets*. In the facets list, find the *Dynamic Web Module* entry and ensure that its version is **3.1**. Press Apply followed by Close.

g. Right-click on the project and choose *New - Folder* to create a folder named **WebContent/WEB-INF**. **Note:** Please ensure that WEB-INF is spelled correctly and is in all uppercase characters.

WebContent is where you will put your application's HTML, JSP and related files and WEB-INF is the standard JEE Web application directory for configuration files.

Right-click on the project and choose *Properties*, then click *Deployment Assembly*. Press the *Add* button, then choose *Folder* and press Next. Select the WebContent folder (not WEB-INF). Press Finish, followed by Apply and OK.

h. Right-click on the project's *Java Resources* folder and choose *New - Source folder* and create a folder named **src** in the **lab06Part** project. This is where you will put your Java code.

i. Right-click on the WebContent folder and choose *New - JSP File* to create a JSP named **index.jsp**. On the second page of the wizard, uncheck the *Use JSP Template* checkbox so the wizard creates an empty file.

j. You can complete the JSP later, but as a quick test to ensure that everything's OK so far, in the *Project Explorer*, right-click on index.jsp and choose *Run As - Run on Server*. On the first wizard page, select your Tomcat server, then press Finish.

Tomcat should start, and Eclipse should display an empty Web page which you can close.

k. Right-click on the project and choose *Properties*, then click *Targeted Runtimes*. Ensure that the Apache Tomcat v8.0 entry has a checkbox.

_7. Next, copy files from the *lab01* project to the new Web project:

a. In the Project Explorer pane, expand the *lab01* 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 *lab06Part1Web/Java Resources* folder, then right-click on the *src* folder and choose *Paste* to copy the Spring bean classes to the new project.

c. Repeat the copy and paste to copy the *log4j.properties* file from lab01 into the new project's *src* folder.

d. Repeat the copy and paste to copy the *spring.xml* file from lab01 into the new project's WebContent/WEB-INF folder.

_8. Next, configure the project for Spring:

a. Generate a Web deployment descriptor. Right-click on the project and choose *Java EE Tools - Generate Deployment Descriptor Stub*.

In the Project Explorer, expand the WebContent/WEB-INF folder to see the generated web.xml.

b. Double-click on the web.xml file file to open it into the editor.

Copy and paste the following text from the **{Lab Installation Directory}/starters/spring/lab06/listener.txt** above the *welcome-file-list* start tag:

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/spring.xml</param-value>
</context-param>
<listener>
    <display-name>ContextLoaderListener</display-name>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

This configures the Web application so that it can access a Spring application context.

Save and close the deployment descriptor.

_9. Next, create a simple servlet that will access the Spring beans:

a. In the Project Explorer, right-click on the *lab06Part1Web* project and choose *New - Servlet* to start the wizard.

b. On the first wizard page, for the *Java package*, enter **servlets**.

For the *Class name*, enter **DisplayFlier**, then press Next.

c. On the next page, examine the defaults, then press Next.

d. On the final page, in the "Which method stubs" section, uncheck the *doPost* box, then press Finish.

Eclipse opens the new servlet into the Java editor.

_10. Complete the new servlet's *doGet* method:

a. Delete any code in doGet() that the wizard generated.

b. Retrieve a reference to the Spring application context:

```
ServletContext servletContext = getServletContext();
WebApplicationContext ctx =
    WebApplicationContextUtils.getRequiredWebApplicationContext(
        servletContext);
```

Choose *Source - Organize Imports* so that Eclipse imports the required types.

c. Look in Flier.java and note that it configures a Spring bean named *flier01*.

Back in the servlet's doGet() method, use the Spring application context to retrieve a reference to the *flier01* bean in the normal Spring fashion:

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

d. In Flier.java, you injected the *flier01* bean with a *ContactInfo* bean. In the servlet's doGet() method, retrieve the ContactInfo bean from the flier:

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

e. Initialize the servlet's HTML output stream:

```
response.setContentType("text/html");
PrintWriter out = response.getWriter();
```

f. Output simple HTML content containing the flier's information:

```
out.println("<html><body>");
out.println("<p>Flier name: " +
   flier01.getFlierName() + "</p>");
out.println("<p>Flier ID: " +
   flier01.getFlierID() + "</p>");
out.println("<p>Flier email: " +
   inf.getEmailAddress() + "</p>");
out.println("</body></html>");
```

g. Import types as necessary, then save the servlet.

_11. To test, in the Project Explorer, right-click on *servlets/DisplayFlier.java* and choose *Run As - Run on Server*, select the Tomcat server, then press Finish. Restart the server if prompted, then wait for the servlet to load - you should eventually see a Web page with the flier info.

_12. In the Eclipse Servers tab, right-click on the Tomcat server and remove all projects from the Tomcat server. Then right-click again and stop the server. Finally, right-click again and choose *Clean Tomcat Work Directory*.

Close all of your editing and browser windows in Eclipse.

Part 2: 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. Copy the last part's project and clean it up a bit.

a. In the Project Explorer, right-click on the lab06Part1 project and choose *Copy*, then right-click on the blank area in the Project Explorer and choose *Paste*, copying the project to **lab06Part2**.

b. In *Java Resources/src*, delete the *servlets* package.

c. In *WebContent/WEB-INF*, delete the *spring.xml* file.

d. Right-click on the project and choose *Properties*, then find the *Web Project Settings* entry. Change the *Context root* to **Lab06Part2**.

_2. In the last part, you configured the project for Spring, but not SpringMVC. You need to do a bit more configuring for SpringMVC:

a. Open WebContent/WEB-INF/web.xml deployment descriptor into the editor.

b. Delete the *context-param* and *listener* elements completely. We will use Java configuration instead of XML.

c. Copying and pasting from **{Lab Installation Directory}/starters/lab06/spring-servlet.txt**, insert the following immediately before the *welcome-file-list* element:

```
<servlet>
    <servlet-name>springmvc</servlet-name>
    <servlet-class>
       org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <init-param>
      <param-name>contextClass</param-name>
      <param-value>
        org.springframework.web.context.support.AnnotationConfigWebApplicationC
      </param-value>
    </init-param>
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>
         com.oaktreeair.config.FrequentFlierConfig
      </param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>springmvc</servlet-name>
  <url-pattern>*.html</url-pattern>
</servlet-mapping>
```

This markup indicates that your application will provide a class named FrequentFlierConfig in the com.oaktreeair.config package that uses Java configuration.

d. Save and close the Web deployment descriptor.

_3. In the last step, you configured the Spring dispatcher servlet to load Java configuration class named *FrequentFlierConfig*. Let's create it:

a. In the Project Explorer, right-click on the *Java Resources/src* folder and create a class named **FrequentFlierConfig** in a new package named **com.oaktreeair.config**.

b. Annotate the class as a Java configuration class, enabling Spring MVC and to scan the *com.oaktreeair.ffprogram* package and subpackages, looking for components:

```
@Configuration
@EnableWebMvc
@ComponentScan("com.oaktreeair.ffprogram")
```

c. Within the class, configure the view resolver:

```
@Bean
public ViewResolver configureViewResolver()
{
    InternalResourceViewResolver viewResolve =
       new InternalResourceViewResolver();
    viewResolve.setPrefix("/WEB-INF/jsp/");
    viewResolve.setSuffix(".jsp");

    return viewResolve;
}
```

This configures the Spring Dispatcher Servlet to look for JSPs in a folder named **/WEB-INF/jsp**.

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

a. In the Project Explorer, right-click on the project's *WebContent/WEB-INF* folder and choose *New - Folder* to create a folder named **jsp** - this is where all of the program's JSPs will reside.

b. Right-click on the new *jsp* folder and choose *New - JSP File* and create a JSP named **findFlier.jsp**. On the second 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:

```
<%@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". Recall that you earlier configured the web.xml for the URL pattern "*.html" - with the combination of these two configurations, the controller will respond to a URL that ends in "/findFlier.html".

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:

```
@RequestMapping(method = RequestMethod.GET)
public void setupForm(Model model)
{
}
```

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:

```
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. To run and test, in the Eclipse Servers tab, right-click on the Tomcat server and add the project to the Tomcat server, then restart Tomcat. Open a Web browser and enter:

```
http://localhost:8080/lab06Part2/findFlier.html
```

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 **flier02** and press Submit - you should see the not-found message.

_8. The application worked, but let's fix up a couple of inconveniences:

a. Open *WebContent/WEB-INF/web.xml* into the editor.

Find the *welcome-file-list* element, and delete all of the entries EXCEPT for *index.jsp*. This page will act as the starting point for the application.

b. Edit WebContent/index.jsp. This will act as the application's *welcome file*.

In the new JSP, the only content should be a scripting element that redirects to your input form via the Spring dispatcher servlet:

```
<%
  response.sendRedirect("findFlier.html");
%>
```

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

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

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

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

_9. To test with the updates, in the Project Explorer, right-click on the *lab06Part2* project itself and choose *Run As - Run on Server* and press Finish. Restart Tomcat if necessary.

Eclipse will run the index.jsp welcome-file, which redirects to the actual start page via the Spring dispatcher servlet.

Note also 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.

_10. Remove all applications from Tomcat and stop the server. Then clean the Tomcat work directory.

Lab 7: 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. If you did NOT do Lab06 Spring MVC, then please execute the steps in Lab06 Part 1 to configure Tomcat.

_2. Next, you need to set up Tomcat so it can access the database you will later create:

   a. Stop the Tomcat server if it's running.

      Using Windows Explorer or My Computer, copy the **{Lab Installation Directory}/lib/h2/h2-1.3.161.jar** file to the **{Tomcat Installation Directory}/lib** folder.

_3. 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.

_4. In the H2 GUI, set the *JDBC URL* to **jdbc:h2:tcp://localhost/~/lab07**, typing this VERY carefully.

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

_5. 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 a starters file in **{Lab Installation Directory}/starters/spring/lab07**):

```
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.

Part 2: Setting Up the Web Project

In this part, you will create a servlet-based Web 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 *lab01* - it doesn't depend on the "experiments". If you did not finish the basic part of lab01, you should either finish it or ask the instructor to help you get that lab's solution.

_2. If necessary, switch to the JEE perspective by choosing *Window - Open Perspective - Other - Java EE (default)*.

_3. In the same fashion as in the last lab, open a command prompt and change to **{Lab Installation Directory}/workspace/lab07**. Examine the provided *build.gradle* script and note that it specifies dependencies for Spring, Spring JDBC and Spring Web.

Run the script to generate an Eclipse project, then import the project into Eclipse. **Note:** Be sure to only import lab07!

_4. After importing the project into Eclipse, continue configuring the project:

a. Right-click on the project and choose *Properties*, then click *Project Facets*. In the facets list, find the *Dynamic Web Module* entry and ensure that its version is **3.1**. Press Apply followed by Close.

b. Right-click on the project and choose *New - Folder* to create a folder named **WebContent/WEB-INF**.

WebContent is where you will put your application's HTML, JSP and related files and WEB-INF is the standard JEE Web application directory for configuration files.

Right-click on the project and choose *Properties*, then click *Deployment Assembly*. Press the *Add* button, then choose *Folder* and press Next. Select the WebContent folder (not WEB-INF). Press Finish, followed by Apply and OK.

c. Right-click on the project's *Java Resources* folder and choose *New - Source folder* and create a folder named **src** in the **lab07** project. This is where you will put your Java code.

d. Right-click on the WebContent folder and choose *New - JSP File* to create a JSP named **index.jsp**. On the second page of the wizard, uncheck the *Use JSP Template* checkbox so the wizard creates an empty file.

e. You can complete the JSP later, but as a quick test to ensure that everything's OK so far, in the *Project Explorer*, right-click on index.jsp and choose *Run As - Run on Server*. On the first wizard page, select your Tomcat server, then press Finish.

Tomcat should start, and Eclipse should display an empty Web page which you can close. You should stop Tomcat as well.

f. Right-click on the project and choose *Properties*, then click *Targeted Runtimes*. Ensure that the Apache Tomcat v8.0 entry has a checkbox.

g. Generate a Web deployment descriptor. Right-click on the project and choose *Java EE Tools - Generate Deployment Descriptor Stub*.

In the Project Explorer, expand the WebContent/WEB-INF folder to see the generated web.xml.

_5. Next, copy files from the *lab01* project to the new Web project:

a. In the Project Explorer pane, expand the *lab01* 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 *lab07/Java Resources* folder, then right-click on the *src* folder and choose *Paste* to copy the Spring bean classes to the new project.

Note: Since you will read the data for the objects from a database instead of having the Spring container initialize them, you can edit the Segment, ContactInfo and Flier classes and remove the Spring @Component and @Value annotations.

c. Repeat the copy and paste to copy the *log4j.properties* file from lab01 into the new project's *src* folder.

_6. Next, you will configure a data source that represents the database. Follow these steps:

a. Create a folder named **META-INF** in the WebContent folder, ensuring that it's spelled correctly and all uppercase.

b. Create a file named **context.xml** in the *WebContent/META-INF* folder.

c. Complete the file, copying from **{Lab Installation Directory/spring/starters/lab07/context.txt** so it looks like:

```
<Context path="/lab07" docBase="lab07" debug="0">
  <Resource name="jdbc/flier"
            auth="Container"
            type="javax.sql.DataSource"
            username="sa" password="sa"
            driverClassName="org.h2.Driver"
            url="jdbc:h2:tcp://localhost/~/lab07"/>
</Context>
```

This defines a Tomcat data source with JNDI name *jdbc/flier*.

_7. Next, create a *resource reference* for the data source in the Web project:

a. In the Project Explorer, double-click on the *WebContent/WEB-INF/web.xml* file to open it into the deployment descriptor editor.

b. Immediately before the *welcome-file-list* element, define the resource reference, copying from **{Lab Installation Directory}/starters/lab07/resource.txt**:

```
<resource-ref>
    <res-ref-name>jdbc/flier</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
    <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
```

c. Save and close the Web deployment descriptor.

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 *Java Resources: src* folder and choose *New - Interface* and create an interface named **SegmentDao** in a new package named **com.oaktreeair.ffprogram.dao**.

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 *JavaResources: src/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*. You will need that ID when you later write servlets.

_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.

Part 4: Web Front End for the DAO

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

Steps:

_1. Configure the data source using Java configuration:

a. In a new **com.oaktreeair.ffprogram.config** package, create a class named **FrequentFlierConfig**.

b. Annotate the class:

```
@Configuration
@ComponentScan("com.oaktreeair.ffprogram")
```

c. Configure a data source bean using a JNDI lookup:

```
@Bean
public DataSource flierDataSource()
{
    JndiDataSourceLookup dsLookup =
        new JndiDataSourceLookup();
    dsLookup.setResourceRef(true);
    DataSource dataSource = dsLookup.getDataSource(
        "jdbc/flier");
   return dataSource;
}
```

Import the *javax.sql.DataSource* type. Note how the JNDI name matches the name you setup in context.xml and web.xml.

d. Open the WEB-INF/web.xml deployment descriptor. Copying and pasting from **{Lab Installation Directory}/starters/lab07/listener.txt**, immediately before the *welcome-file-list* element, configure Spring to reference your configuration class:

```
<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>

<context-param>
  <param-name>contextClass</param-name>
    <param-value>
     org.springframework.web.context.support.AnnotationConfigWebApplicationCont
    </param-value>
</context-param>

<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    com.oaktreeair.ffprogram.config.FrequentFlierConfig
  </param-value>
</context-param>
```

_2. Next, create a simple servlet that will access the segment count via the DAO:

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

b. On the first wizard page, for the *Java package*, enter **com.oaktreeair.ffprogram.servlets**.

For the *Class name*, enter **DisplaySegmentCount**, then press Next.

c. On the next page, examine the defaults, then press Next.

d. On the final page, in the "Which method stubs" section, uncheck the *doPost* box, then press Finish.

Eclipse opens the new servlet into the Java editor.

_3. Complete the new servlet's *doGet* method:

a. Delete any code generated by the wizard.

b. Retrieve a reference to the Spring application context:

```
ServletContext servletContext = getServletContext();
WebApplicationContext ctx =
    WebApplicationContextUtils.getRequiredWebApplicationContext(
        servletContext);
```

Choose *Source - Organize Imports* so that Eclipse imports the required types.

c. Retrieve a reference to the DAO from the application context in the normal fashion:

```
SegmentDao dao = . . .;
```

d. Initialize the servlet's HTML output stream:

```
response.setContentType("text/html");
PrintWriter out = response.getWriter();
```

e. Output simple HTML content containing the segment count:

```
out.println("<html><body>");
out.println("<p>Segment count: " +
   dao.getSegmentCount() + "</p>");
out.println("</body></html>");
```

f. Import types as necessary, then save the servlet.

_4. To test, in the Project Explorer, right-click on *DisplaySegmentCount.java* and choose *Run As - Run on Server*, select Tomcat, then press Finish. Restart Tomcat if prompted, then wait for the server to start - you should see a Web page with the segment count (4).

Part 5: Completing the Application

In this part, you will complete the DAO and write servlets 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 *JavaResources: src/com.oaktreeair.ffprogram.dao/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:

```
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. In the same fashion as earlier, create a new servlet named **DisplayAllSegments** whose doGet() method uses the DAO to retrieve the Segment list and then displays them. For extra credit (!), display the list in an HTML table.

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

_9. In the Servers tab, right-click on the Tomcat server and remove the project from the server, then right-click again and stop the server. Then clean the Tomcat work directory.

Lab 8: Spring AOP

In this lab, you will write a simple program that uses Spring AOP.

Objectives:

- To work with Spring AOP to centralize a cross-cutting concern.

Part 1: Spring AOP

Steps:

_1. This lab depends on successfully completing the basic parts of the first Spring lab. If you did not finish the basic part of the first lab, you should either finish it or ask the instructor to help you get the first lab's solution.

_2. Follow this procedure to make a copy of your lab01 project:

   a. Choose *File - Close All* to close all open editors.

   b. In the Project Explorer, right-click on the *lab01* project and choose *Copy*.

   c. Right-click the blank, white area of the Project Explorer and choose *Paste*.

      In the *Copy Project* dialog, enter **lab08** and press OK.

_3. We need to update the project's dependencies so we can use the AOP annotations and classes:

   a. Open the build.gradle file into the Eclipse editor.

      Add the following dependencies, then save and close build.gradle.

```
compile 'org.springframework:spring-aop:4.2.6.RELEASE'
compile 'org.springframework:spring-aspects:4.2.6.RELEASE'
```

   b. Open a command prompt window and change to the project's directory. If your computer has the standard lab setup:

```
cd \Users\username\springclass\workspace\lab08
```

   c. Regenerate the Eclipse project:

```
gradle eclipse
```

   d. Back in Eclipse, right-click on the project and choose *Refresh*.

_4. Review the code in the com.oaktreeair.ffprogram package and especially note the BonusCalc interface and the BonusCalcImpl class. Your job in this lab is to use an aspect to measure the performance of the two methods in this service.

   Before writing the aspect, add a "sleep" call in the two methods in BonusCalcImpl to simulate a long-running method so that we actually have something to measure:

```
try
{
    double d = Math.random();
    int time = (int)(d * 1000);
    TimeUnit.MILLISECONDS.sleep(time);
}
catch (InterruptedException e) {}
```

_5. Your next job is to write the performance-measuring aspect.

a. In the *com.oaktreeair.ffprogram.util* package, create a class named **PerformanceAspect**.

b. Annotate the class with @Aspect and @Component.

c. As described in the student notes, define an empty method named **myPointcut** that will act as a *pointcut* name. Annotate the method so that the point intercepts all calls to the *calcBonus* method, regardless of arguments.

d. Write another method in the aspect named **measure** with this signature:

```
public Object measure(ProceedingJoinPoint joinPoint)
  throws Throwable
{

}
```

e. Annotate the *measure* method to reference your pointcut:

```
@Around("myPointcut()")
```

f. Complete the *measure* method so that it notes the current time before and after the call to the intercepted method and logs it to the console:

```
long startTime = System.currentTimeMillis();
Object o = joinPoint.proceed();
long endTime = System.currentTimeMillis();
System.out.println("calcBonus elapsed time: " +
   (endTime - startTime) + " milliseconds");
return o;
```

_6. In the spring.xml file, add the **aop:aspectj-autoproxy** element as described in the course notes.

_7. Run the FrequentFlierProgram - you should see the elapsed time.

Lab 9: 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\springclass\workspace\lab09
   ```

   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
   ```

   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 clean 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 *lab09* 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 *lab09* project looking in the *Java Resources:src/com.oaktreeair.ffprogram* and *com.oaktreeair.ffprogram.dao* packages. Figure 1 summarizes the application:

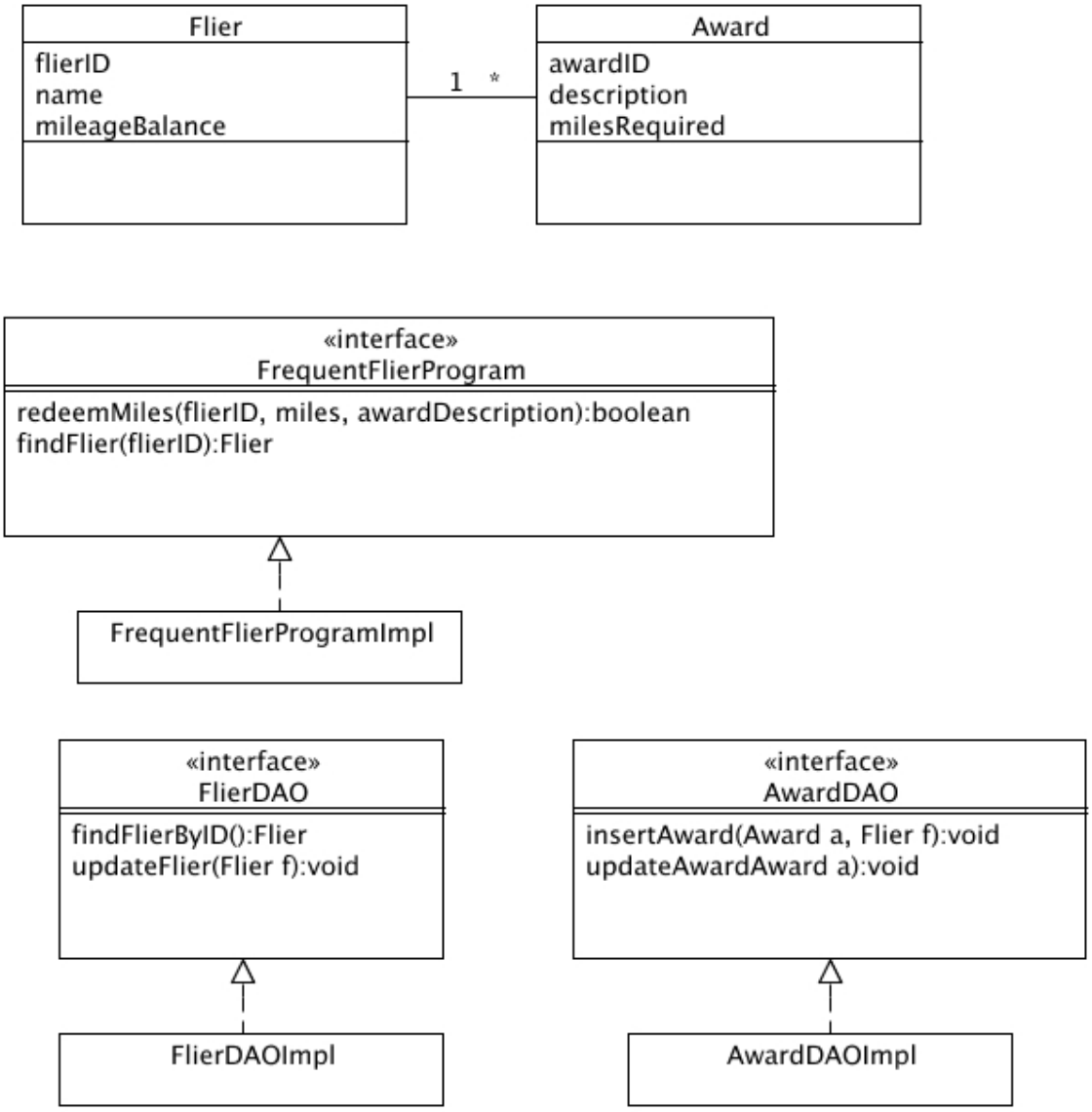

Figure 1: Application Model

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.

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/~/lab09**, then enter **sa** for both username and password, then press Connect.

d. Back in Eclipse, expand the *WebContent/WEB-INF* folder, 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));
```

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 - you will configure the data source that's injected later:

```
@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:

```
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.

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 - you will configure the data source that's injected later:

```
@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* package that implements the *FrequentFlierProgram* interface.

b. Annotate the class with **@Service**, assigning a name of **frequentFlierProgram**.

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:

```
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;
```

g. Save FrequentFlierProgramImpl.java.

Part 4: Configuring Spring

In this part, you will configure Spring to use transactions and to inject dependencies.

Steps:

_1. Next, configure the datasource and transaction manager using Java configuration:

a. In a new **com.oaktreeair.ffprogram.config** package, create a class named **FrequentFlierConfig**.

b. Annotate the class:

```
@Configuration
@ComponentScan("com.oaktreeair.ffprogram")
```

c. Configure a data source bean using a JNDI lookup:

```
@Bean
public DataSource flierDataSource()
{
    JndiDataSourceLookup dsLookup =
        new JndiDataSourceLookup();
    dsLookup.setResourceRef(true);
    DataSource dataSource = dsLookup.getDataSource(
        "jdbc/flier");
   return dataSource;
}
```

Import the *javax.sql.DataSource* type. Note how the JNDI name matches the name defined in the provided META-INF/context.xml and WEB-INF/web.xml files. Examine those files and ensure you understand them.

d. Using the example in the book as a guide, write another @Bean method to configure a DataSourceTransactionManager that references the datasource you configured in the flierDataSource() method.

_2. Next, reference your Java config file from web.xml:

a. Open the WEB-INF/web.xml deployment descriptor. Copying and pasting from **{Lab Installation Directory}/starters/lab07/listener.txt**, immediately before the *welcome-file-list* element, configure Spring to reference your configuration class:

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    com.oaktreeair.ffprogram.config.FrequentFlierConfig
  </param-value>
</context-param>
```

Note: Don't copy the entire contents of listener.txt - only the *context-param* element shown above.

Part 5: Testing

In this part, you will run a provided servlet that redeems miles for an Award on a Flier's account.

Steps:

_1. Open the *Java Resources: src/com.oaktreeair.ffprogram.servlets/AwardServlet.java* servlet and ensure that you understand it. Note how it uses the Spring context to retrieve a reference to the FrequentFlierProgram bean, then attempts to redeem a mileage award, displaying the results to a simple Web page.

_2. Test the program by right-clicking on AwardServlet.java and choosing *Run As - Run on Server*. After the server starts, you should see the "before" and "after" of the attempt to redeem miles.

_3. Copying and pasting from **{Lab Installation Directory}/starters/spring/lab09/display-awards.txt**, insert this code into the servlet immediately following the first *out.println* that displays the mileage balance:

```
Collection<Award> awards = f.getAwards();
out.println("<table border='1'>");
for(Award a : awards)
{
    out.println("<tr>");
    out.println("<td>" + a.getDescription() + "</td>");
    out.println("<td>" + a.getMilesRequired() + "</td>");
    out.println("</tr>");
}
out.println("</table>");
```

Also paste this code after the second *out.println* that displays the mileage balance, modifying it as necessary to get it to compile.

Try running the servlet again to see the Award info.

Optional Lab 10: Testing

In this optional lab, you will write tests for a Spring application.

Objectives:

- To write tests for Spring applications

Part 1: Introduction to JUnit

In this part, you will write unit tests for a Spring application, testing outside of the Spring container.

We will provide you with a starter project that has the code for which you will write tests.

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\springclass\workspace\lab10
      ```

   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
      ```

      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 clean 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 *lab10* 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 *lab10* project in the *Project Explorer* window pane by clicking on the "+" symbol to expand it. Note the following:

- The *src/main/java* folder contains an *insurance* package with the code under test and an *insurance.config* package containing a Spring Java configuration class.

_4. Expand the *insurance* package and note the following provided classes:

- **Customer.java** represents a customer of an insurance company.
- **Claim.java** represents a claim on an insurance policy.
- **Policy.java** represents an insurance policy. It references a Customer and a collection of Claims and has logic to calculate the total and average claim amounts.
- **InsuranceApp.java** is the entry point of the actual application.

_5. Let's start by creating a separate *source* folder in which you will write your test cases. This technique lets you have multiple packages with the same name in the separate source folder. That's helpful, since it lets your test cases be in the same package as the code under test, but yet be separate on the file system so it's easy to deploy the code under test and exclude the test cases.

In the Package Explorer, right-click on the *lab10* project and choose *New - Source Folder* and name the source folder **src/test/java**.

_6. Next, create your test-case class:

a. Right-click on the *src/test/java* folder and choose *New - Other - Java - JUnit - JUnit Test Case* to start the wizard.

b. On the first wizard page, select the *New JUnit 4 test* option.

For the *package*, enter **insurance**.

For the *Name*, enter **TestPolicy**.

For the *Class under test*, press the Browse button and start typing **Policy** then choose the *Policy - insurance - lab10/src/main/java* entry and press OK.

Press Next.

c. On the next page, select three of the methods in the *Policy* class: addClaim, getTotalClaims and getAverageClaim, then press Finish.

OK any prompt to add JUnit 4 to the build path.

Eclipse generates the test-case class and brings its source into the Java editor.

_7. Examine the generated test-case class:

- It imports the necessary types for JUnit 4 test-cases.
- It defines a POJO (plain old Java object) class for the test-case class.
- It defines testXXXX() methods for each of the three methods you selected in the wizard. Note that each of these test-case methods has the *@Test* annotation and that the wizard generated a call to *fail()* in each test-case method.

_8. Now let's try running the test-cases as is:

a. In the Package Explorer, right-click on TestPolicy.java and choose *Run As - JUnit test*.

Eclipse runs the tests. Note that the results show three failures out of three tests run. That should make sense, because each of the test-case methods has a fail() call in it.

_9. Next, let's start writing a test case:

a. In the *testAddClaim* method, delete the *fail()* call.

b. Still in testAddClaim, test this method's basic function:

```
Customer cust01 = new Customer();
Policy policy01 = new Policy(cust01);
Claim claim01 = new Claim();
policy01.addClaim(claim01);
assertEquals("Claim count should be 1", 1,
        policy01.getClaimCount());
```

Save to compile.

c. Run the JUnit test again. You should now have only two failures.

_10. Now let's test the getTotalClaims method:

a. In *getTotalClaims()*, delete the call to *fail()*.

b. Add the following to testTotalClaims. Note that you can copy and paste some of this from *testAddClaim* that you did in the last step.

```
Customer cust01 = new Customer();
Policy policy01 = new Policy(cust01);
Claim claim01 = new Claim();
claim01.setClaimID(12);
claim01.setAmount(5000);
policy01.addClaim(claim01);
Claim claim02 = new Claim();
claim02.setClaimID(44);
claim02.setAmount(7000);
policy01.addClaim(claim02);
assertEquals("Total claims should be 12000",
        12000.0, policy01.getTotalClaims(), 0.0);
```

The last parameter on assertEquals is the "delta" of rounding error your test will tolerate.

c. Run the JUnit test again. You should now have only one failure.

_11. Next, let's clean up the test-case class to reduce redundancy. To do this, you will write a "set up" method, and in it, create a *test fixture* that has an exam, questions and results. Follow these steps:

a. In TestPolicy.java, write a field that holds a reference to an Policy:

```
private Policy policy01;
```

b. Write a new method in TestPolicy.java:

```
@Before
public void setUp()
{
}
```

After saving, choose *Source - Organize Imports* to import *org.junit.Before*.

c. CUT all of the code from the testGetTotalClaims method EXCEPT the assertion to the clipboard, then paste it into the setUp method.

   In the setUp method, modify the *new Policy(cust01)* line so that it stores the returned Policy in the field (remove the declaration of the local Policy variable).

d. Delete ALL of the code from testAddClaim EXCEPT the assertion.

e. Run the test again - testGetTotalClaims should work, but testAddClaim now fails. Why?

f. In the testAddClaim method, edit assertEquals so that the expected value is two instead of one (the setUp method adds two Claims).

g. Run the JUnit tests as before. You should have one failure out of the three runs.

_12. Using the techniques you learned in the chapter and in this lab, complete the testGetAverageClaim method to test the average exam score that you'd expect given the data in the test fixture.

After completing this part, you should have a green bar indicating that all of the test cases passed.

Part 2: Integration Testing with Spring

In this part, you will use Spring dependency injection to build the test fixture.

Steps:

_1. Create a new testing class that uses the AbstractJUnit4SpringContextTests superclass:

a. Right-click on the *src/test/java/insurance* package and choose *New - Class* to start the wizard.

b. For the *Package*, confirm **insurance**.

   For the *Name*, enter **TestPolicySpring**.

   For the *Superclass*, press the Browse button and start typing **AbstractJU**, then select *AbstractJUnit4SpringContextTests* and press OK followed by Finish.

c. Open the TestPolicy.java class from the last part, then copy the Policy field, the setUp method and all of the testing methods into the clipboard.

   Paste the text into the TestPolicySpring.java class.

_2. Now you will update the TestPolicySpring class to use Spring/JUnit integration:

a. Above the *public class TestPolicySpring* declaration, write an annotation that points to the Spring configuration class:

```
@ContextConfiguration(
   classes=insurance.config.InsuranceConfig.class)
```

   You will need to import the ContextConfiguration type.

b. In the Project Explorer, right-click on TestPolicySpring.java and choose *Run As - JUnit Test* - the test should work as in the last lab with a green bar.

c. Even though the tests worked, they are building the test fixture manually. Edit the setUp method and replace the code with:

```
policy01 = (Policy)applicationContext.getBean("policy01");
```

d. Then examine the provided *src/main/java/insurance.config/InsuranceConfig* class and note that it creates several beans and initializes them using dependency injection. Update your tests as necessary so they reflect this configuration. (Hint: use the "delta" parameter in the assertEquals method.)

e. Run TestPolicySpring as a JUnit test - it should work as before.

Note that you might need to switch Eclipse back to the JUnit view window from the Console to see the results of running the tests.

_3. There is one more thing you can do to simplify the test:

a. Above the *private Policy policy01* field, write an **Autowired** annotation. Import the type.

b. Comment out or delete the entire setUp method.

c. Run the test as before. Spring can "autowire" the policy field since there's only one Policy bean configured in spring.xml.

Note that you might need to switch Eclipse back to the JUnit view window from the Console to see the results of running the tests.

Lab 11: 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

In this part, you will create your RESTful service.

To save time, we will provide you with a starter project.

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\springclass\workspace\lab11
      ```

   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, noting that it specifies a runtime dependency for the Jackson JSON library:

      ```
      write build.gradle
      ```

      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 clean 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 *lab11* 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 imported project:

- There is a *src/main/java/com.oaktreeair.ffprogam* package that includes the domain model for the frequent-flier application.
- The *WEB-INF/web.xml* file configures the Spring MVC dispatcher servlet with a URL pattern of /. It also configures the *contextConfigLocation* to reference a configuration class named *FFProgramConfig* that you will write later.

_4. Next, let's write a Web application context listener that will create an in-memory "database" of Segment objects and store them at application scope:

a. Right-click on the *src/main/java* folder and create a new class in a new package, *com.oaktreeair.ffprogram.listeners* named **MyContextListener** that implements *ServletContextListener* from the *javax.servlet* package.

b. Annotate the class with **@WebListener** - this registers the listener.

c. In the two generated methods, change the names of the arguments to **sce** instead of *arg0*.

d. In the *contextInitialized* method (NOT in contextDestroyed), write the following to create a Segment list and store it at application scope:

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

Segment s = new Segment();
s.setFlightNumber(5079);
s.setMiles(345);
s.setOriginatingCity("CID");
s.setSegmentDate(new Date());
s.setSegmentNumber((long)12);
segments.add(s);

s = new Segment();
s.setFlightNumber(2486);
s.setMiles(2165);
s.setOriginatingCity("SEA");
s.setSegmentDate(new Date());
s.setSegmentNumber((long)13);
segments.add(s);

s = new Segment();
s.setFlightNumber(1431);
s.setMiles(1431);
s.setOriginatingCity("ORD");
s.setSegmentDate(new Date());
s.setSegmentNumber((long)14);
segments.add(s);

sce.getServletContext().setAttribute("segments", segments);
```

_5. Annotate the Segment class with **@XmlRootElement**. This will enable Spring to transform Segment objects to/from XML.

_6. Right-click on the *src/main/java* folder and create a new class in a new package, *com.oaktreeair.ffprogram.controllers*, named **SegmentController**.

Complete the new class:

a. Annotate the class so that it's a Spring MVC controller and then annotate it with a request-mapping URL of **rest**.

b. Define a field and inject a reference to the servlet context where your in-memory "database" of Segments resides:

```
@Autowired
private ServletContext servletCtx;
```

c. Within the controller, write a nested class named **ResourceNotFoundException** that extends RuntimeException. Annotate the exception class with **@ResponseStatus**. Your code should look like:

```
@ResponseStatus(value = HttpStatus.NOT_FOUND)
private class ResourceNotFoundException
   extends RuntimeException
{
}
```

d. Write a the skeleton of a service method named **getSegment** that accepts the segment number as a parameter and returns a Segment.

Annotate the method so that it responds to a URL ending in **/segment/{segmentNumber}** for an HTTP GET request. Annotate the method's parameter with **@PathVariable**. Also annotate the Segment return value with **@ResponseBody**.

Code the method so that it looks up the Segment in the in-memory list and returns its reference. You can retrieve a reference to the segment list like:

```
List<Segment> segments =
   (List<Segment>)servletCtx.getAttribute("segments");
```

If there's no Segment with the specified segment number, throw the ResourceNotFoundException.

_7. Next, use Java configuration to configure your application:

a. In a new package, *com.oaktreeair.ffprogram.config*, create a class named **FFProgramConfig**, extending *WebMvcConfigurerAdapter*.

b. Annotate the class:

```
@Configuration
@EnableWebMvc
@ComponentScan("com.oaktreeair.ffprogram.controllers")
```

c. Override the superclass's *configureDefaultServletHandling* method, and in the overriden method, enable serving resources:

```
@Override
public void configureDefaultServletHandling(
        DefaultServletHandlerConfigurer configurer)
{
    configurer.enable();
}
```

_8. In the Servers tab, right-click on the server and add the project to the server. Restart the server.

_9. To test, open a Web browser and enter the following URL:

```
http://localhost:8080/lab11/rest/segment/12
```

You should see an XML document that has information about Segment #12. Note that you may need to view the page source in the browser to see the actual XML.

Try again with a URL for Segment #59 - you should get a Not Found response.

If you have cURL installed (you can download it from http://curl.haxx.se/download.html), try using it to set the Accept header to **application/json** and retrieve the JSON representation of Segment #12:

```
curl -v -H "Accept: application/json"
    http://localhost:8080/lab11/rest/segment/12
```

Note: Enter the above into a command prompt all on a single line. We have split it here so it fits on the printed page.

Optional Challenge Part 2: Implementing CRUD and Using a jQuery Ajax Client

In this part, you will finish the service so it provides the following:

- Support for retrieving all segments.
- 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 run Web pages that use jQuery Ajax to exercise the service.

Steps:

_1. Update the service by writing an additional four methods that GET all segments, CREATE a new segment, UPDATE an existing segment and DELETE a segment.

_2. We have provided you with several HTML files that use the jQuery JavaScript library to issue Ajax requests to your service. Run them, then examine them to see if you can make sense of how they work. You can find them in the *starters/lab11* folder.

Test with cURL and/or a browser.