Advanced Spring

Lab Exercises

Eclipse Mars/Tomcat 8 Version

Version 1.0.2 June 22, 2016



Copyright © Descriptor Systems, 2016
All Rights Reserved.

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

All trademarks belong to their respective companies.

You can contact Descriptor Systems at:

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

Advanced Spring Labs

Lab 1: Introduction to Spring Boot

Part 1: A Simple Spring Application

Part 2: A Spring Boot Web Application

Lab 2: Additional Spring Boot Topics

Part 1: Working with Starter Dependencies

Part 2: Working with Autoconfiguration and Deployment

Part 3: Using an External Database

Part 4: Deploying as a WAR

Lab 3: Spring and JMS

Part 1: Creating the Project

Part 2: Creating the Messaging Layer

Part 3: Creating the Web Layer

Lab 4: Spring Batch

Part 1: A Simple Spring Batch Application

Part 2: Processing Input and Generating Output

Part 3: Custom Item Reader and Writer

Lab 5: Introduction to Spring Integration

Part 1: A Hello, World Spring Integration Application

Part 2: Using a Messaging Gateway

Lab 6: Spring Integration Messages and Channels

Part 1: Using Point-to-Point Channels

Part 2: Adding a Publish-Subscribe Channel

Lab 7: Spring Integration Endpoints

Part 1: Project and Database Setup

Part 2: Building the Application

Lab 8: Introduction to Spring XD

Part 1: Getting Started with Spring XD
Lab 9: Spring XD Modules
Part 1: Creating a Stream with Modules

Lab 1: Introduction to Spring Boot

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

Objectives:

- To get started with Spring Boot

Part 1: A Simple Spring Application

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

Steps:

_1. Ask your instructor for the following information:

Lab Installation directory: _____

Eclipse Installation directory: _____

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

```
Lab Installation directory:       c:\Users\username\advspring
Eclipse Installation Directory:   c:\Users\username\java\eclipse
```

Substitute your actual username for the placeholder "username".

If your computer is running Mac OS X, the directories will be something like:

```
Lab Installation directory:       /Users/username/advspring
Eclipse Installation Directory:   /Applications
```

Substitute your actual username for the placeholder "username".

_2. Your first job is to create the starter Spring Boot application. You will use the Spring Boot **Command Line Interface (CLI)** to generate the starter. Follow these instructions:

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

```
spring --version
```

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

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

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
cd \Users\username\advspring\workspace\lab01
```

On Mac OS X:

```
cd /Users/username/advspring/workspace/lab01
```

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

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

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

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

_3. Use your Windows Wordpad to open **build.gradle**:

```
write build.gradle
```

Change the *baseName* from *demo* to **lab01**, then delete the entire *eclipse{ }* block at the bottom of the script.

Note: Wordpad works best for these script files if you click the *View* menu and set *Word wrap* to *No wrap*.

_4. Build the project, installing Gradle. In your command prompt or Terminal:

Windows: Enter:

```
gradlew
```

Mac OS X: Enter:

```
chmod +x gradle
./gradlew
```

_5. In the command prompt or Terminal, generate an Eclipse project:

Windows:

```
gradlew eclipse
```

Mac OS X:

```
./gradlew  eclipse
```

_6. **Windows Instructions:**

Start Eclipse running:

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

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

This directory should be something like:

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

Note: Be sure to substitute your computer's actual username for the "username" above.

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

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

This directory should be something like:

```
c:\Users\username\advspring\workspace
```

Note: Be sure to substitute your computer's actual username for the "username" above.

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

We also recommend running Eclipse as a maximized window.

_7. **Mac OS X Instructions:**

Start Eclipse running:

a. Use Finder to navigate to the **{Eclipse Installation Directory}**.

This directory should be something like:

```
/Applications/eclipse
```

b. Start Eclipse running by double-clicking on *Eclipse.app*.

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

This directory should be something like:

```
/Users/username/advspring/workspace
```

Note: Be sure to substitute your computer's actual username for the "username" above.

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

We also recommend running Eclipse as a maximized window.

_8. Eclipse supports the notion of *perspectives*, which are sets of window panes that let you work in a particular role. For this lab, you will work in the *Java* perspective, which contains window panes that let you create and edit Java programs.

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

_9. Import the project:

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

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

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

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

d. Note that in the Project list, *lab01* is selected. Press Finish to complete the import.

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

_10. Next, create the (small) business model for this application:

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

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

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

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

_11. Create the "controller" that will act as the program's starting point:

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

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

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

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

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

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

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

_12. Run the program:

Windows: At the command prompt, enter:

```
gradlew bootRun
```

Mac OS X: In the Terminal, enter:

```
./gradlew bootRun
```

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

_13. Get rid of the Spring Boot banner and run from within Eclipse:

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

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

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

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

Part 2: A Spring Boot Web Application

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

Steps:

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

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

Windows:

```
cd ..\lab01p2
```

Mac OS X:

```
cd ../lab01p2
```

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

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

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

_2. Use your Windows Wordpad to open **build.gradle** and change the *baseName* from *demo* to **lab01p2**. Then delete the entire *eclipse{...}* block at the bottom.

```
write build.gradle
```

_3. In the same fashion as in the first part of lab, in the command prompt or Terminal, generate an Eclipse project.

_4. In the same fashion as in the first part of lab, import the existing project into Eclipse.

_5. In Eclipse, copy and paste the *com.example.shipping* package from the *lab01/src/main/java* package to the equivalent location in the new project.

_6. Create the view pages for the application:

a. Open the *src/main/resources* folder then right-click on the *templates* folder and choose *New - Other - Web - HTML File* and create a file named **shipping.html**.

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Modify the content so it looks something like:

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>Calculate Domestic Shipping</title>
</head>
<body>
 <h1>Calculate Shipping</h1>
 <form method="post" action="/shipping">
  <p>
   Zip code: <input type="text" name="zip"></input>
  </p>
  <p>
   <input type="Submit" value="Calculate"></input>
     </p>
 </form>
</body>
</html>
```

b. Create another HMTL file in the same folder named **shippingresult.html** with the following content:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Shipping Result</title>
  </head>
  <body>
    <h1>Shipping Result</h1>
    <p>
      The cost to ship to
      <span th:text="${zip}"></span> is
      <span th:text="${shipping}"></span>.
    </p>
  </body>
</html>
```

Note that we use *Thymeleaf* expressions to display data from Spring beans.

_7. Under *src/main/java*, create a new package named **com.example.service** and in it a new class named **ShippingController**. This class doesn't need to implement any interfaces.

Complete the controller:

a. Annotate the class:

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

Note that */shipping* matches the *action* of the HTML form you created in an earlier step.

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

b. Auto-wire a field of type *Shipping*.

c. Write a method that displays the input form in shipping.jsp:

```
@RequestMapping(method = RequestMethod.GET)
public String setup()
{
    return "shipping";
}
```

d. Write a method that responds to the form submission and then forwards to the 'result' Thymeleaf HTML view page:

```
@RequestMapping(method = RequestMethod.POST)
public String submitForm(@RequestParam String zip, Model model)
{
    double cost = shipping.calculateDomestic(zip);

    model.addAttribute("zip", zip);
    model.addAttribute("shipping", cost);

    return "shippingresult";
}
```

Note how the model attributes match the Thymeleaf expressions, ${...} in the shippingresult.html you created earlier.

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

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

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

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

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

_9. Test the application:

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

b. Open a browser and enter this URL:

```
http://localhost:8080/shipping
```

You should see your application's input form. Enter any zip code you wish, then submit the form. You should see the results of your shipping calculation.

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

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 2: Additional Spring Boot Topics

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

Objectives:

- To customize Spring boot starter dependencies
- To customize Spring boot autoconfiguration
- To deploy to a WAR

Part 1: Working with Starter Dependencies

In this part, you will create and run a Spring Boot Web application that uses a database.

Steps:

_1. Ask your instructor for the following information:

Lab Installation directory: _____

Eclipse Installation directory: _____

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

```
Lab Installation directory:       c:\Users\username\advspring
Eclipse Installation Directory:   c:\Users\username\java\eclipse
```

Substitute your actual username for the placeholder "username".

If your computer is running Mac OS X, the directories will be something like:

```
Lab Installation directory:       /Users/username/advspring
Eclipse Installation Directory:   /Applications
```

Substitute your actual username for the placeholder "username".

_2. Open a command-prompt or Terminal and change to **{Lab Installation Directory}/lab02**.

_3. Use the following command to browse the starter dependencies available to the Spring Boot CLI:

```
spring init --list | more
```

Hint: Use the space bar to advance pages in the output.

Note especially the **h2** (H2 database), **jdbc** (Spring JDBC support), **web** (Spring MVC) and **thymeleaf** starters.

_4. Use the Spring Boot CLI to generate a starter project with these specifications:

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
Dependencies:          web, h2, jdbc, thymeleaf
Build language:        Gradle
Options:               Expanded project
Spring Boot version:   1.3.1.RELEASE
```

_5. Use Windows WordPad to cdit the *build.gradle* and note the starter dependencies. Especially note how there's no version numbers - Spring Boot will find versions that work well together.

In the *jar* task configuration, change the *baseName* to **lab02** and delete the entire **eclipse{...}** block at the end. Be sure to save the file.

_6. Now let's examine which versions of the dependencies (and THEIR dependencies) that Spring Boot will bring in. In your command prompt or Terminal:

a. **Windows**:

```
gradlew dependencies
```

Mac OS X:

```
./gradlew dependencies
```

This will first build the project and then show a large tree of the libraries and their versions.

b. Find the section titled **runtime - Runtime classpath for source set 'main'**, and identity the versions of:

`org.springframework:spring-jdbc:` _____

`com.fasterxml.jackson.core:jackson-databind:` _____

`com.h2database:h2:` _____

`org.yaml:snakeyaml:` _____

_7. Note how the starter for Web automatically brought in the Jackson library, which converts Java objects to/from JSON. In our application, however, we will not use JSON, Jackson isn't really necessary, so let's exclude it:

a. In build.gradle, in the *dependencies* configuration block, modify the starter-web dependency so it looks like:

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
compile('org.springframework.boot:spring-boot-starter-web') {
   exclude group: 'com.fasterxml.jackson.core'
}
```

b. Repeat the above step for the *thymeleaf* starter dependency.

c. Display the dependency tree again - the Jackson dependency should be gone.

Hint: You can filter the tree display using the *find* or *grep* commands:

Windows:

```
gradlew dependencies | find "jackson"
```

Mac OS X:

```
./gradlew dependencies | grep "jackson"
```

d. Repeat the process to exclude the *hibernate-validator* dependency. **Note:** The *group* name is **org.hibernate** (you can also exclude by *module* - the module name is **hibernate-validator**).

_8. Now let's imagine that our QA team has identified a critical bug for our application in the H2 database, and recommends that we use a specific version. Let's modify the build to use that version:

a. Since we have an explicit dependency on H2, we can directly specify the version number. Modify the H2 dependency so it looks like:

```
runtime('com.h2database:h2:1.0.60')
```

b. Display the dependency tree again - it should show the required H2 version.

_9. Next, lets specify a specific version of a *transitive* dependency. YAML provides an alternate syntax for specifying application properties, and the *jdbc* starter brings it in so we can configure database connections using YAML syntax via the snakeyaml library.

Our scenario is that we want to use a different, older version. To accomplish that in Gradle, we need to first exclude the suggested (newer) version, then explicitly specify the version we want:

a. Modify the *starter-jdbc* dependency so it looks like:

```
compile('org.springframework.boot:spring-boot-starter-jdbc') {
  exclude module: 'snakeyaml'
}
```

b. Add the following:

```
compile('org.yaml:snakeyaml:1.8')
```

c. Display the dependency tree - it should have the 1.8 version of snakeyaml after an arrow '->' listing the suggested version.

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

_10. Now lets build the Java domain classes for the application, which will display a list of customers.

a. Generate an Eclipse project and bring the existing project into Eclipse.

b. In a new *com.example.domain* package, create a class named **Customer** with the following fields.

```
private int customerID;
private String name;
private String email;
private String phone;
```

Generate get/set methods and a toString() method.

_11. Next, create the data layer for the application:

a. In a new *com.example.dao* package, create a new interface named **CustomerDao**.

Add the following methods:

```
List<Customer> allCustomers();
```

b. Create a class named **CustomerDaoImpl** that implements the interface.

c. Annotate the class with **@Repository**.

d. Within the class, outside of the methods, write an inner class that maps JDBC rows to Customer objects (**Note**: You can copy and paste this from **{Lab Installation Directory/starters/lab02/starters.txt**):

```
private class CustomerRowMapper implements RowMapper<Customer>
{
    public Customer mapRow(ResultSet rs, int rowNum) throws SQLException
    {
        Customer cust = new Customer();

        cust.setCustomerID(rs.getInt("CustomerID"));
        cust.setEmail(rs.getString("Email"));
        cust.setName(rs.getString("Name"));
        cust.setPhone(rs.getString("Phone"));

        return cust;
    }
}
```

e. After the inner class, define fields in the DAO class:

```
private JdbcTemplate template;
private CustomerRowMapper mapper;
private DataSource customerDataSource;
```

Note: The *import* for the DataSource type should be:

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
import javax.sql.DataSource;
```

f. After the fields, inject the data source:

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

g. Complete the allCustomers() method:

```
@Override
public List<Customer> allCustomers()
{
    List<Customer> customers = template
            .query("SELECT * FROM Customer", mapper);

    return customers;
}
```

_12. Next, create the Web layer:

a. In a new *com.example.controllers* package, create a class named **CustomerController**. Annotate the class so it's a Spring MVC controller responding to requests whose URL ends in **/customers**.

b. Inject a DAO reference:

```
private CustomerDao dao;

@Autowired
public CustomerController(CustomerDao dao)
{
    this.dao = dao;
}
```

c. Write a method that sets up the customer list for display:

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
@RequestMapping(method = RequestMethod.GET)
public String allCustomers(Model model)
{
    List<Customer> custList = dao.allCustomers();
    if (custList != null)
    {
        model.addAttribute("customers", custList);
    }
    return "customers";
}
```

_13. Next, create the Web user interface:

a. Right-click on the *src/main/resources/template* folder and create an HTML file named **customers.html** with the following content:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Customer List</title>
    <link rel="stylesheet" href="/style.css"></link>
  </head>
  <body>
    <h1>Customer List</h1>
    <table>
      <tr>
        <th>Customer ID</th>
        <th>Name</th>
        <th>Phone</th>
        <th>Email</th>
      </tr>
        <tr th:each="cust : ${customers}">
          <td th:text="${cust.customerID}">ID</td>
          <td th:text="${cust.name}">Name</td>
          <td th:text="${cust.phone}">Phone</td>
          <td th:text="${cust.email}">Email</td>
        </tr>
    </table>
  </body>
</html>
```

b. In the *src/main/resources/static* folder, create a file named **style.css** with the following very simple CSS styling:

```
td, th { border: 1px solid; }
```

_14. Edit the *build.gradle* file and modify it:

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

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

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

b. In the *dependencies* block, add:

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

_15. Next, create an SQL script that will create the Customer table and populate it with a couple of rows. In the *src/main/resources* folder, create a file named **schema.sql** with the following contents:

```
CREATE TABLE customer(customerID INT IDENTITY PRIMARY KEY,
  name VARCHAR(100), phone VARCHAR(100), email VARCHAR(100));

INSERT INTO customer (name, phone, email) VALUES
  ('Sam Smith', '555-1212', 'sam@sam.com');
INSERT INTO customer (name, phone, email) VALUES
  ('Sue Jones', '555-1313', 'sue@sue.com');
```

_16. Test the application:

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

b. Open a browser and enter this URL:

```
http://localhost:8080/customers
```

You should see your application's table of customer info with two rows of data.

Part 2: Working with Autoconfiguration and Deployment

In this part, you will rewrite the Customer list application so it provides the customer list as a RESTful resource.

Steps:

_1. Open a command-prompt or Terminal and change to **{Lab Installation Directory}/lab02p2**.

_2. Use the Spring Boot CLI to generate a starter project with these specification:

```
Dependencies:         web, h2, jdbc
Build language:       Gradle
Options:              Expanded project
Spring Boot version:  1.3.1.RELEASE
```

_3. Edit the *build.gradle* and modify it:

a. In the *jar* task configuration, change the *baseName* to **lab02p2**.

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

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

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

c. In the *dependencies* block, add:

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

d. Delete the **eclipse{...}** block at the end.

_4. Generate an Eclipse project and import it into Eclipse.

_5. Copy the following from the *lab02* project to equivalent locations in the new project:

- The *com.example.domain* package
- The *com.example.dao* package
- The *src/main/resources/schema.sql* file
- The *src/main/static/resources/static/style.css* file

_6. Next, create a controller that provides the customer list as a RESTful resource:

a. In a new *com.example.controllers* package, create a class named **CustomerController**.

b. Annotate the class so that it's a controller.

c. In the same fashion as in the last part of lab, autowire a reference to the DAO.

d. Write a method that returns all customers as a RESTful resource with URL */customer*:

```
@RequestMapping(value = "/customer",
  method = RequestMethod.GET)
public @ResponseBody
List<Customer> getSegments()
{
    return dao.allCustomers();
}
```

e. Write a nested class that serves up 404 NOT FOUND errors:

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

_7. As a quick test, in your command prompt or Terminal, use *gradlew bootRun* to start the app, then open a browser and navigate to:

```
http://localhost:8080/customer
```

You should see the customer list in JSON format.

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Stop the server.

_8. Next, let's write a HTML page that uses JavaScript to retrieve and display the customer list. For convenience, we will use the jQuery JavaScript library. Follow these steps:

a. In the *src/main/resources/static* folder, create a file named **customers.html** with the following content (copy from the starters file):

```
<!DOCTYPE html>
<html>
  <head>
    <title>Customer List</title>
    <script
   src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.3/jquery.min.js">
    </script>
    <script src="scripts.js">
    </script>
    <link rel="stylesheet" href="/style.css"></link>
  </head>
  <body>
     <h1>Customer List</h1>
     <table id="custlist">
      <tr>
        <th>Customer ID</th>
        <th>Name</th>
        <th>Phone</th>
        <th>Email</th>
      </tr>

     </table>
  </body>
</html>
```

b. In the *src/main/resources/static* folder, create a file named *scripts.js* with the following content:

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
$(document).ready(function()
{
 $.getJSON("customer", function(theList)
 {
     for(var i = 0; i < theList.length; i++)
     {
         var row =
             "<tr>" +
             "<td>" + theList[i].customerID + "</td>" +
             "<td>" + theList[i].name + "</td>" +
             "<td>" + theList[i].phone + "</td>" +
             "<td>" + theList[i].email + "</td>" +
             "</tr>";

         $('#custlist').append(row);
     }
 });
});
```

_9. Use *gradlew bootRun* to run the app, then browse to:

```
http://localhost:8080/customers.html
```

You should see the customer list.

Part 3: Using an External Database

So far, your applications have used the H2 database in its in-memory mode, which is the default for Spring Boot. This mode is very fast for development, but doesn't actually save data persistently (Spring Boot ran the schema.sql script automatically each time you booted the app).

In this part, you will convert the customer-list application to use an external database.

Steps:

_1. You must configure the database for your program. We will use the free, open-source HypersonicSQL database to hold the customer information. Follow these steps to configure the database:

Windows Instructions:

a. Open a command-prompt window (choose Start - Run and type **cmd**), then change to the **{Lab Installation Directory}/lib** directory. If your computer has the standard lab setup:

```
c:
cd \Users\username\advspring\lib
```

Substitute your actual user name, e.g. *libtrain1* for the "username" above.

b. Start the HypersonicSQL server running:

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
java -cp hsqldb-2.3.3.jar org.hsqldb.Server -database mydb
```

The server will start and display a sign-on message. If prompted by Windows, allow access for the server.

Note: Be sure to leave this command-prompt running until you are finished with the lab.

c. Open another command-prompt window and change to same directory as you did in the last step.

d. Start the HypersonicSQL administrator's GUI:

```
java -cp hsqldb-2.3.3.jar org.hsqldb.util.DatabaseManager
```

e. In the GUI, change the *Type* to **HSQL Database Engine Server**, then press OK. **Warning:** Be sure to choose the Server option!

f. In the GUI, in the command area between the Clear and Execute buttons, create and populate the database table, by entering the following (copy and paste from the 'starters' file):

```
CREATE TABLE customer(customerID INT IDENTITY PRIMARY KEY,
  name VARCHAR(100), phone VARCHAR(100), email VARCHAR(100));
```

Then press the Execute button.

Press the Clear button, then execute:

```
INSERT INTO customer (name, phone, email) VALUES
  ('Sam Smith', '555-1212', 'sam@sam.com');
INSERT INTO customer (name, phone, email) VALUES
  ('Sue Jones', '555-1313', 'sue@sue.com');
INSERT INTO customer (name, phone, email) VALUES
  ('Harry Wolfe', '555-1414', 'harry@wolfe.com');
```

To test the new table, press the CLEAR button, then enter:

```
SELECT * from customer
```

Press the Execute button -- you should see the customer info.

You can close the GUI before continuing, but be sure to leave the server running.

Mac OS X Instructions:

a. In the Finder, open your /Applications/Utilities folder, then start Terminal. In the Terminal, change to the **{Lab Installation Directory}/lib** directory. If your computer has the standard lab setup:

```
cd /Users/username/advspring/lib
```

Substitute your actual user name, e.g. *libtrain1* for the "username" above.

b. Start the HypersonicSQL server running:

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
java -cp hsqldb-2.3.3.jar org.hsqldb.Server -database mydb
```

The server will start and display a sign-on message.

Note: Be sure to leave this Terminal running until you are finished with the lab.

c. Open another Terminal and change to same directory as you did in the last step.

d. Start the HypersonicSQL administrator's GUI:

```
java -cp hsqldb-2.3.3.jar org.hsqldb.util.DatabaseManager
```

e. In the GUI, change the *Type* to **HSQL Database Engine Server**, then press OK. **Warning:** Be sure to choose the Server option!

f. In the GUI, in the command area between the Clear and Execute buttons, create and populate the database table, by entering the following (copy and paste from the 'starters' file):

```
CREATE TABLE customer(customerID INT IDENTITY PRIMARY KEY,
  name VARCHAR(100), phone VARCHAR(100), email VARCHAR(100));
```

Then press the Execute button.

Press the Clear button, then execute:

```
INSERT INTO customer (name, phone, email) VALUES
  ('Sam Smith', '555-1212', 'sam@sam.com');
INSERT INTO customer (name, phone, email) VALUES
  ('Sue Jones', '555-1313', 'sue@sue.com');
INSERT INTO customer (name, phone, email) VALUES
  ('Harry Wolfe', '555-1414', 'harry@wolfe.com');
```

To test the new table, press the CLEAR button, then enter:

```
SELECT * from customer
```

Press the Execute button -- you should see the customer info.

You can close the GUI before continuing, but be sure to leave the server running.

_2. Use Windows Explorer or the Mac OS X Finder to copy the *lab02p2* directory into the *workspace* folder with a new name of **lab02p3**. Then import the *lab02p3* project into Eclipse.

_3. In Eclipse, edit *build.gradle* and remove the H2 dependency line. Then add a dependency to the HSQLDB database:

```
runtime('org.hsqldb:hsqldb')
```

_4. Edit *src/main/resources/application.properties* and add the following lines to override the database autoconfiguration so that your application use the HSQLDB database instead of the in-memory H2 database:

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
spring.datasource.url=jdbc:hsqldb:hsql://localhost/
spring.datasource.username=SA
spring.datasource.password=
spring.datasource.driver-class-name=org.hsqldb.jdbcDriver
```

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

_5. Delete the *src/main/resources/schema.sql* file - we don't need it since we created and populated the table manually.

_6. Open a command-prompt or Terminal and change to **{Lab Installation Directory}/lab02p3**.

_7. Use *gradlew clean bootRun* to run the app, then browse to:

```
http://localhost:8080/customers.html
```

You should see the customer list, now getting its data from the external HSQLDB database.

Part 4: Deploying as a WAR

So far, you have tested your applications using Spring Boot's embedded Tomcat. In this part, you will create a Web archive (WAR) file and deploy it to a standalone Tomcat instance.

Steps:

_1. Use Windows Explorer or the Mac OS X Finder to copy the *lab02p3* directory into the *workspace* folder with a new name of **lab02p4**. Then import the *lab02p4* project into Eclipse.

_2. In Eclipse, edit *build.gradle* and change the *jar* configuration block to a **war** block (just change the 'j' to a 'w').

_3. Edit *customers.html* and ensure that there's no leading slash in the *href* for the *style.css* stylesheet.

_4. Since we will run the application as a WAR, not as an executable JAR, we need to ensure that the Spring MVC *dispatcher servlet* is registered with Tomcat. One way to do that is to create a subclass of SpringBootServletInitializer (another would be to create a web.xml file and register the servlet there):

a. In the *com.example* package, create a class named **MyServletInitializer** that's a subclass of *org.springframework.boot.context.web.SpringBootServletInitializer*.

b. Use Eclipse's *Source - Override/Implement Methods* menu to override the *configure()* method.

c. Modify the configure() method so it looks like:

```
@Override
protected SpringApplicationBuilder configure(
        SpringApplicationBuilder builder)
{
    return builder.sources(DemoApplication.class);
}
```

_5. Open a command-prompt or Terminal and change to **{Lab Installation Directory}/lab02p4**.

_6. Use the *gradlew clean build* command to build the WAR, the

Look in the *build/libs* folder - you should see the WAR file.

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

_7. Next, install a standalone Tomcat server and deploy the WAR:

a. Use Windows Explorer or the Mac OS X Finder to unzip **{Lab Installation Directory}/tomcat/apache-tomcat-8.0.27.zip**.

We recommend that you unzip it in your home directory to that same *tomcat* folder, but in any case, note where you unzipped it as the **Tomcat Installation Directory}**:

Tomcat Installation Directory: _____

b. Use Windows Explorer or Mac OS X Finder to copy *lab02p2-0.0.1-SNAPSHOT.war* to **{Tomcat Installation Directory}/webapps**. This deploys the application.

Change the name of the copied file to **lab02p4.war**. Tomcat uses the WAR name as the application's *context root*, which is part of the app's URL.

c. Use Windows Explorer or Mac OS X Finder to copy the **{Lab Installation Directory}/lib/hsqldb-2.3.3.jar** file to the **{Tomcat Installation Directory}/lib** folder. This makes the HSQLDB driver available to Tomcat.

_8. Ensure that the HSQLDB server is still running and if not, start it as described in the last part of lab.

_9. Now you can start Tomcat:

Windows Instructions:

a. Open a command prompt and change to **{Tomcat Installation Directory}** directory.

b. Start Tomcat:

```
bin\catalina.bat run
```

Tomcat should start and in its console, show your app's deployment.

Mac OS X Instructions:

a. Open a Terminal and change to **{Tomcat Installation Directory}** directory.

b. Start Tomcat:

```
bin/catalina.sh run
```

Tomcat should start and in its console, show your app's deployment.

_10. Test your application by opening a browser and navigating to:

```
http://localhost:8080/lab02p4/customers.html
```

You should see the customer list.

Be sure to kill the external Tomcat instance before continuing.

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 3: Spring and JMS

In this lab, you will write an application that uses Spring JMS to send and receive messages.

Objectives:

- To use Spring JMS

Part 1: Creating the Project

In this part, you will create a Spring Boot project that uses Spring JMS. To make things simple, you will use the open-source HornetQ JMS provider in its embedded mode.

Steps:

_1. Open a command-prompt or Terminal and change to **{Lab Installation Directory}/lab03**.

_2. Use the Spring Boot CLI to generate a starter project with these specifications:

```
Dependencies:          web, hornetq, thymeleaf
Build language:        Gradle
Options:               Expanded project
Spring Boot version:1.3.1.RELEASE
```

_3. Edit the *build.gradle* and modify it:

a. In the *jar* task configuration, change the *baseName* to **lab03**.

b. In the *repository* list (not in *buildscript*), add:

```
maven{ url 'http://repository.jboss.org/nexus/content/groups/public'}
```

This enables gradle to fetch a required dependency for running HornetQ (org.jboss.naming:jnpserver).

c. In the *dependencies* block, add:

```
compile('org.hornetq:hornetq-jms-server')
```

d. Delete the **eclipse{...}** block at the end.

_4. Generate an Eclipse project and import it into Eclipse.

_5. Edit *application.properties* and configure HornetQ, defining a queue named **poQueue**:

```
spring.hornetq.mode=embedded
spring.hornetq.embedded.enabled=true
spring.hornetq.embedded.queues=poQueue
```

Note: Please ensure there are no trailing spaces at the end of these lines.

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

_6. In the command prompt or Terminal, use *bootRun* to start the application and ensure that there's no errors. You should see a diagnostic message about deploying poQueue.

Stop the server for now.

_7. In the *com.example.domain* package, create a class named **PurchaseOrder** that implements java.io.Serializable.

Complete the class:

a. Define fields:

```
private int poNumber;
private Date poDate;
private double amount;
```

b. Generate get/set methods.

c. Generate a toString() method.

Part 2: Creating the Messaging Layer

Steps:

_1. In the *com.example.messaging* package, create a class named **SendPurchaseOrder**.

Complete the class:

a. Annotate the class with **@Component**.

b. Define a field of type *JmsTemplate* and annotate it with **@Autowired**.

c. Create a nested class that acts as message creator:

```
private class MyMessageCreator implements MessageCreator
{
    private PurchaseOrder po;

    public MyMessageCreator(PurchaseOrder po)
    {
        this.po = po;
    }

    @Override
    public Message createMessage(Session session) throws JMSException
    {
        ObjectMessage msg = session.createObjectMessage(po);

        return msg;
    }
}
```

d. Write a method named **sendPO** that returns void and accepts an int for the PO number and a double for the amount.

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

In the method, create a PurchaseOrder object and initialize its properties (you can use today's date). Then use the jmsTemplate to send a message to the queue you configured earlier:

```
MyMessageCreator mmc = new MyMessageCreator(po);
jmsTemplate.send("poQueue", mmc);
```

_2. Next, create the message receiver. In the *com.example.messaging* package, create a class named **ReceivePurchaseOrder** that implements *javax.jms.MessageListener*.

Complete the class:

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

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

b. In the onMessage() method, first cast the input parameter to *javax.jms.ObjectMessage*, then call its getObject() method to retrieve the PurchaseOrder object.

Then log the purchase-order number, amount and date at info() level to the logger.

Note that you will need a try/catch block.

_3. Next, create a class that uses Spring's *Java configuration* to configure the application:

a. In the *com.example* package, create a class named **MyConfiguration**.

b. Annotate the class with @Configuration.

c. Define a field of type *javax.jms.ConnectionFactory* and annotate it with @Autowired.

d. Define a configuration method that returns a configured DefaultMessageListenerContainer:

```
@Bean
public DefaultMessageListenerContainer messageListener()
{
    DefaultMessageListenerContainer container =
            new DefaultMessageListenerContainer();
    container.setConnectionFactory(this.connectionFactory);
    container.setDestinationName("poQueue");
    container.setMessageListener(new ReceivePurchaseOrder());
    return container;
}
```

_4. In your command prompt or Terminal, use *bootRun* to start the server and ensure that there's no errors. Then stop the server for now.

Part 3: Creating the Web Layer

Steps:

_1. Create the view pages for the application:

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

a. Right-click on the *src/main/resources/templates* folder and choose *New - Other - Web - HTML File* and create an HTML file named **po.html**.

Modify the content so it looks something like:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Send Purchase Order</title>
  </head>
  <body>
     <h1>Send Purchase Order</h1>
     <form method="post" action="/sendpo">
       <p>
         PO Number: <input type="text" name="number"></input>
       </p>
       <p>
         Amount: <input type="text" name="amount"></input>
       </p>
       <p>
         <input type="Submit" value="Send"></input>
       </p>
     </form>
  </body>
</html>
```

b. Create another HTML file in the same folder named **sendresult.html** with the following content:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Send Result</title>
  </head>
  <body>
    <h1>Send Result</h1>
    <p>
      Message sent. See the application's
      console for details.
    </p>
  </body>
</html>
```

_2. In the com.example.controllers package, create a new class named **PoController**.

Complete the controller:

a. Annotate the class:

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

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

Note that */sendpo* matches the *action* of the HTML form you created in an earlier step.

b. Auto-wire a field of type *SendPurchaseOrder*.

c. Write a method that displays the input form in po.jsp:

```
@RequestMapping(method = RequestMethod.GET)
public String setup()
{
    return "po";
}
```

d. Write a method that responds to the form submission and then forwards to the 'result' JSP:

```
@RequestMapping(method = RequestMethod.POST)
public String submitForm(@RequestParam int number,
        double amount, Model model)
{
    sender.sendPO(number, amount);

    return "sendresult";
}
```

_3. Edit the *build.gradle* file and modify it:

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

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

b. In the *dependencies* block, add:

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

_4. In your command prompt or Terminal, use *bootRun* to start the application. Then browse to *http://localhost:8080/sendpo* - you should see your input form. Enter a PO number and amount, then press Send.

Look in the command prompt or Terminal - you should see a message with the PO information.

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 4: Spring Batch

In this lab, you will write an application that uses Spring Batch to process various kinds of data.

Objectives:

- To use Spring Batch

Part 1: A Simple Spring Batch Application

Steps:

_1. Open a command-prompt or Terminal and change to **{Lab Installation Directory}/lab04**.

_2. Use the Spring Boot CLI to generate a starter project with these specifications:

```
Dependencies:       batch, hsql
Build language:     Gradle
Options:            Expanded project
```

_3. Edit the *build.gradle* and modify it:

a. In the *jar* task configuration, change the *baseName* to **lab04**.

b. Delete the *eclipse{...}* block at the end.

_4. Generate an Eclipse project and import it into Eclipse.

_5. In the *com.example* package, create a class named **BatchConfiguration**, then complete it:

a. Annotate the class:

```
@Configuration
@EnableBatchProcessing
```

The @Configuration annotation is from core Spring and indicates this class provides Java configuration.

The @EnableBatchProcessing annotation enables Spring Batch and sets up a base configuration, including a JobRepository, JobLauncher, JobRegistry and job and step factories. You can then autowire these into your program.

b. Define two fields, one of type **JobBuilderFactory**, named **jobBuilderFactory** the other **StepBuilderFactory** with a similar name. Use @Autowired on each. These two objects let you create jobs and steps, respectively. Note how the @EnableBatchProcessing annotation on the class configures these objects.

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

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


Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

_6. Continue completing the BatchConfiguration class by writing a method that configures your little application's one and only step:

a. Start by writing the method's skeleton:

```
@Bean
public Step step1()
{

}
```

b. In the method, use an inner class to define a Tasklet that contains the step's code:

```
Tasklet task = new Tasklet()
{
    @Override
    public RepeatStatus execute(StepContribution contribution,
            ChunkContext chunkContext) throws Exception
    {
        log.info("STEP ONE!");
        return RepeatStatus.FINISHED;
    }
};
```

Tasklets are a unit of work within a step. This tasklet simply writes to the console. Most real-world steps will do something more interesting, such as reading/writing files or databases.

c. Still in the method, use the injected StepBuilderFactory to create a StepBuilder, assigning a name of "step1", then assign the tasklet and retrieve a TaskletStepBuilder:

```
StepBuilder sb = stepBuilderFactory.get("step1");
TaskletStepBuilder tsb = sb.tasklet(task);
```

d. Use the TaskletStepBuilder to create the step object and return it:

```
return tsb.build();
```

_7. Write a method that configures your application's Job:

a. Start by writing the method's skeleton:

```
@Bean
public Job job() throws Exception
{

}
```

b. Create an *incrementer* object that assigns simple sequence IDs to the job executions. This would be important if the job failed and needed to be restarted:

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
RunIdIncrementer incrementer = new RunIdIncrementer();
```

c. Use the injected JobBuilderFactory to create a JobBuilder, assigning the name "job1". Then assign the incrementer to to JobBuilder:

```
JobBuilder jb = jobBuilderFactory.get("job1");
jb.incrementer(incrementer);
```

d. Create a SimpleJobBuilder, assigning the step you created earlier:

```
SimpleJobBuilder sjb = jb.start(step1());
```

e. Create the Job object and return it:

```
return sjb.build();
```

_8. Test your batch program. In your command prompt or Terminal, use *bootRun* to run - you should see your step's output.

_9. Now, let's add another step:

a. In BatchConfiguration.java, write another method that creates another step that outputs "STEP TWO!".

b. Modify the job() method so that immediately before returning, it calls *next()* to configure the second step:

```
 sjb.next(step2());
```

c. Execute the program - you should see the output from both steps.

Part 2: Processing Input and Generating Output

In this part, you will write a Spring Batch application that reads comma-separated-value files and writes to a database.

Steps:

_1. In a command prompt or Terminal, change to the *lab04p2* directory, then create a project using these specifications:

```
Dependencies:       batch, hsql
Build language:     Gradle
Options:            Expanded project
```

Then generate an Eclipse project and import it into Eclipse.

_2. Edit build.gradle and add the following dependency:

```
compile ('commons-dbcp:commons-dbcp:1.4')
```


Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

We need this so that the data source will create later uses the Apache Commons DBCP for connection pooling.

_3. Expand the *Referenced Libraries* folder, then find the Spring batch core library - its name will be something like *spring-batch-core-3.0.5.RELEASE.jar*. Expand that entry and find *org.springframework.batch.core* and expand it. Figure One shows what it should look like so far:

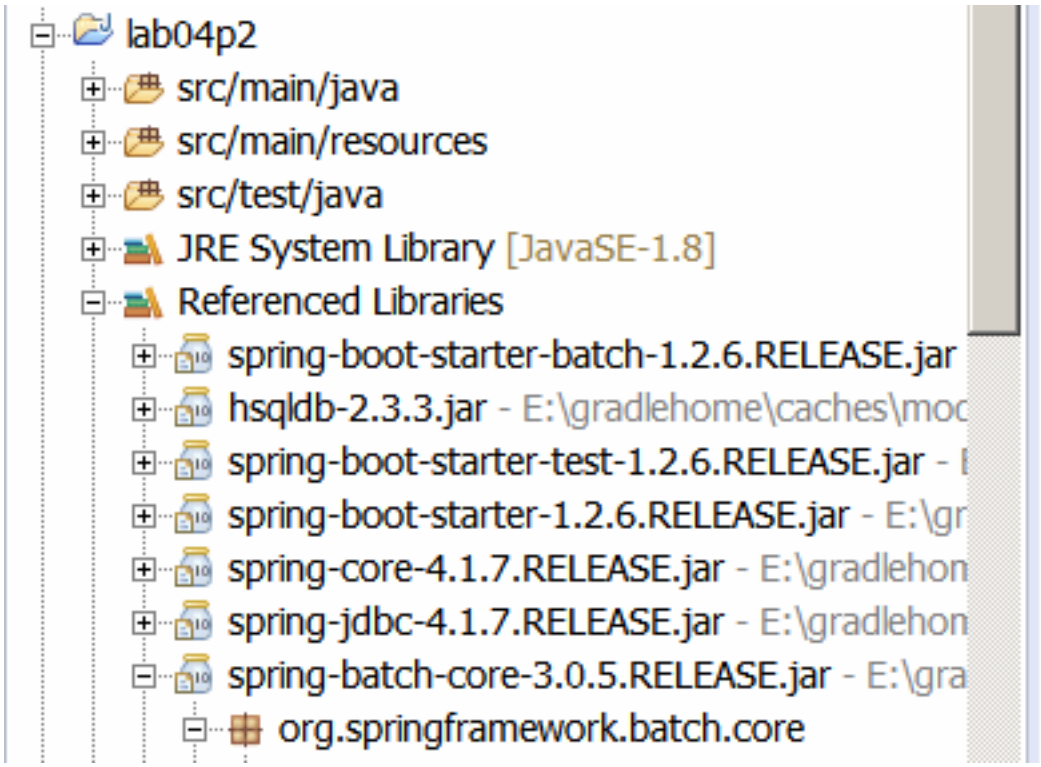


Figure One: Referenced Libraries

Scroll down to find *schema-hsqldb.sql* and double-click to open. You will use this file in a moment - it defines the tables required to store the JobRepository's metadata.

_4. You must configure the database for your program. We will use the free, open-source HypersonicSQL database to hold bank account information. Follow these steps to configure the database:

Windows Instructions:

a. Open a command-prompt window (choose Start - Run and type **cmd**), then change to the **{Lab Installation Directory}/lib** directory. If your computer has the standard lab setup:

```
c:
cd \Users\username\advspring\lib
```

Substitute your actual user name, e.g. *libtrain1* for the "username" above.

b. Start the HypersonicSQL server running:

```
java -cp hsqldb-2.3.3.jar org.hsqldb.Server -database mydb
```


Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

The server will start and display a sign-on message. If prompted by Windows, allow access for the server.

Note: Be sure to leave this command-prompt running until you are finished with the lab.

c. Open another command-prompt window and change to same directory as you did in the last step.

d. Start the HypersonicSQL administrator's GUI:

```
java -cp hsqldb-2.3.3.jar org.hsqldb.util.DatabaseManager
```

e. In the GUI, change the *Type* to **HSQL Database Engine Server**, then press OK. **Warning:** Be sure to choose the Server option!

f. In the GUI, in the command area between the Clear and Execute buttons, create and populate the database table, by entering the following (copy and paste from the 'starters' file):

```
CREATE TABLE account(accountID INT IDENTITY PRIMARY KEY,
   balance DOUBLE, interestRate DOUBLE, customerID INT);
```

Then press the Execute button.

g. Back in Eclipse, select and copy all of the contents of *schema-hsqldb.sql* into the clipboard.

In the HSQL GUI, press the Clear button, then paste the clipboard into the command area. Press Execute to run the script.

From the menu, choose *View - Refresh Tree* to see the tables you just created.

Mac OS X Instructions:

a. In the Finder, open your /Applications/Utilities folder, then start Terminal. In the Terminal, change to the **{Lab Installation Directory}/lib** directory. If your computer has the standard lab setup:

```
cd /Users/username/advspring/lib
```

Substitute your actual user name, e.g. *libtrain1* for the "username" above.

b. Start the HypersonicSQL server running:

```
java -cp hsqldb-2.3.3.jar org.hsqldb.Server -database mydb
```

The server will start and display a sign-on message.

Note: Be sure to leave this Terminal running until you are finished with the lab.

c. Open another Terminal and change to same directory as you did in the last step.

d. Start the HypersonicSQL administrator's GUI:

```
java -cp hsqldb-2.3.3.jar org.hsqldb.util.DatabaseManager
```

e. In the GUI, change the *Type* to **HSQL Database Engine Server**, then press OK. **Warning:** Be sure to choose the Server option!

f. In the GUI, in the command area between the Clear and Execute buttons, create and populate the database table, by entering the following (copy and paste from the 'starters' file):

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
CREATE TABLE account(accountID INT IDENTITY PRIMARY KEY,
  balance DOUBLE, interestRate DOUBLE, customerID INT);
```

Then press the Execute button.

g. Back in Eclipse, select and copy all of the contents of *schema-hsqldb.sql* into the clipboard.

In the HSQL GUI, press the Clear button, then paste the clipboard into the command area. Press Execute to run the script.

From the menu, choose *View - Refresh Tree* to see the tables you just created.

_5. Back in Eclipse, create a file containing the CSV content for an Account:

a. Right-click on the project and create a file named **accounts.csv**. If Windows asks to find a program to open the file, press Cancel. If Eclipse launches Microsoft Excel, close it. If Mac OS X opens the file in a spreadsheet program or TextEdit, close it.

In Eclipse, right-click on the file and choose *Open With - Text Editor*. Then add the following:

```
12,1000.00,0.03,1
13,2000.00,0.02,2
14,3000.00,0.04,0
```

This defines data for three accounts.

b. In the Package Explorer, right-click on the accounts.csv file and choose *Properties*. In the properties, note the file's *Location*. You will need it later, so note it here:

CSV file path: _____

_6. Still in Eclipse, edit the *application.properties* file to disable Spring Batch from automatically running the database scripts:

```
spring.batch.initializer.enabled=false
```

_7. In a new *com.example.domain* package, create a class named **Account** and in it, write fields and get/set methods that correspond to the ACCOUNT database table you created earlier.

Also generate a toString() method.

_8. In the same fashion as the last part of lab, create a class *com.example.BatchConfiguration*, annotating it appropriately.

Then like before, auto-wire fields for the JobBuilderFactory and StepBuilderFactory.

_9. In the BatchConfiguration class, define a bean-configuration method to reference your database:

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
@Bean
public DataSource dataSource()
{
    DataSourceBuilder dsb = DataSourceBuilder.create();
    dsb.url("jdbc:hsqldb:hsql://localhost/");
    dsb.driverClassName("org.hsqldb.jdbcDriver");
    dsb.username("SA");
    dsb.password("");

    return dsb.build();
}
```

_10. In the BatchConfiguration class, define a bean-configuration method that processes lines from the CSV file:

a. Write the method's skeleton:

```
@Bean
public LineMapper<Account> lineMapper()
{

}
```

b. Create a tokenizer for the Account CSV:

```
DelimitedLineTokenizer tokenizer =
        new DelimitedLineTokenizer();
tokenizer.setDelimiter(",");
tokenizer.setNames(new String[]{"accountID",
        "balance","interestRate","customerID"});
```

Note: The Strings in the array should match the names of the properties in your Account domain class. If your Account class uses different property names than the ones shown here, please change the values in this array to match.

c. Create an object that maps to your Account class:

```
BeanWrapperFieldSetMapper<Account> fieldSetMapper =
        new BeanWrapperFieldSetMapper<Account>();
fieldSetMapper.setTargetType(Account.class);
```

d. Create and configure the line mapper and return it:

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
DefaultLineMapper<Account> lineMapper =
        new DefaultLineMapper<Account>();
lineMapper.setLineTokenizer(tokenizer);
lineMapper.setFieldSetMapper(fieldSetMapper);
return lineMapper;
```

_11. Create a bean-configuration method that actually processes the CSV file using the line-mapper:

a. First, create the method's skeleton:

```
@Bean
public ItemReader<Account> csvFileReader()
{

}
```

b. Define a Resource that references the CSV file:

```
Resource res = new FileSystemResource("xxxxx");
```

For the 'xxxxx', use the account.csv file path that you noted earlier. **Hint for Windows users:** Ensure that the path separators in the path string are forward slashes, not back slashes.

c. Create and configure the ItemReader to process the CSV file, using the Spring Batch FlatFileItemReader type:

```
FlatFileItemReader<Account> itemReader =
        new FlatFileItemReader<Account>();
itemReader.setLineMapper(lineMapper());
itemReader.setResource(res);
return itemReader;
```

_12. Create a bean-configuration method to write the database:

a. Write the method's skeleton:

```
@Bean
public ItemWriter<Account> jdbcItemWriter()
{

}
```

b. Define the SQL INSERT statement:

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
final String sql =
    "INSERT INTO account " +
    "(balance,interestRate,customerID) " +
    "VALUES(:balance,:interestRate,:customerID)";
```

c. Create and configure the ItemWriter, using the Spring Batch JdbcBatchItemWriter type, then return it:

```
JdbcBatchItemWriter<Account> itemWriter =
        new JdbcBatchItemWriter<Account>();
itemWriter.setDataSource(dataSource());
itemWriter.setSql(sql);

itemWriter.setItemSqlParameterSourceProvider(
        new BeanPropertyItemSqlParameterSourceProvider
           <Account>());
return itemWriter;
```

_13. Create a bean-configuration method for your batch's one and only step:

a. First, write the method's skeleton:

```
@Bean
public Step step1()
{


}
```

b. First, create the StepBuilder:

```
StepBuilder sb = stepBuilderFactory.get("step1");
```

c. Then create a 'chunked' SimpleStepBuilder with a chunk size of three. This will read three rows from the CSV file and then do a single JDBC command, batching the three INSERT commands:

```
SimpleStepBuilder<Account, Account> ssb =
        sb.chunk(3);
```

d. Set the ItemReader and ItemWriter on the SimpleStepBuilder:

```
ssb.reader(csvFileReader());
ssb.writer(jdbcItemWriter());
```

e. Build and return the configured Step object:

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
return ssb.build();
```

_14. Finally, configure the batch Job itself:

a. Write the method's skeleton:

```
@Bean
public Job insertIntoDbFromCsvJob()
{

}
```

b. Create an incrementer:

```
RunIdIncrementer incrementer = new RunIdIncrementer();
```

c. Use the injected JobBuilderFactory to create a JobBuilder, assigning the name "job1". Then assign the incrementer to to JobBuilder:

```
JobBuilder jb = jobBuilderFactory.get("job1");
jb.incrementer(incrementer);
```

d. Create a SimpleJobBuilder, assigning the step you created earlier:

```
SimpleJobBuilder sjb = jb.start(step1());
```

e. Create the Job object and return it:

```
return sjb.build();
```

_15. Use *gradlew bootRun* to run the application.

_16. Back in the HSQL GUI, enter:

```
SELECT * FROM account
```

Press Execute - you should see rows corresponding to the data in the CSV file.

Press the Clear button, then enter:

```
SELECT * FROM batch_job_execution
```

Press Execute - you should see rows corresponding to the job execution, including the start and end times as well as the exit codes.

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Part 3: Custom Item Reader and Writer

In this part of lab, you will write a Spring Batch application that uses a custom ItemReader to process a text file containing information about a sporting event. The file will contain three lines of text for each event. You will write a simple custom ItemWriter that simply outputs to the console.

Note: In this part, we deliberately give you fewer step-by-step instructions. At this point in the class, we assume you know what you are doing!

Steps:

_1. In your command prompt or Terminal change to the *lab04p3* directory and use the Spring Boot CLI to create an expanded Gradle project with *batch* and *hsql* dependencies.

Generate an Eclipse project and import it into Eclipse.

_2. Right-click on the project and create a file named **games.txt**, then enter the following information:

```
2014-10-23
Iowa
Nebraska
2014-10-30
Michigan
Ohio State
2014-11-7
Penn State
Maryland
```

This file contains data for three games, each game taking up three lines: the game date and the two team names.

Note: Ensure that there are no blank lines at the beginning or end of the file.

_3. In a new *com.example.domain* package, create a class named **Game** with String fields for *gameDate*, *team1Name* and *team2Name*. Generate get/set methods and a toString() method.

_4. In the *com.example* package, create a class named **MyItemReader** that implements both *ItemReader* and *ItemStream* from the *org.springframework.batch.item* package. **Note**: ItemReader is a generic interface - you will need to parameterize it with Game.

Complete the class:

a. Define a field of type *BufferedReader*.

b. In the *open()* method, use a FileReader to open your games.txt file, specifying the full path. Then create a BufferedReader, passing the FileReader. Store the BufferedReader reference in the field.

If a FileNotFoundException occurs, rethrow it as an ItemStreamException.

c. In the *close* method, close the BufferedReader, rethrowing ItemStreamException if something goes wrong.

d. In the *read()* method, read three lines using the BufferedReader: the game date, team1Name and team2name. Check to see if any are null - if not, create a Game object and call its set() methods. Then return the Game reference.

If any of the lines are null, return null.

_5. In the *com.example* package, create a class named **MyItemWriter** that implements *ItemWriter* from the *org.springframework.batch.item* package. **Note:** ItemWriter is a generic interface - you will need to parameterize it with Game.

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Then complete the class:

a. Define a field of type *org.apache.log4j.Logger* for ItemWriter.class.

b. In the *write()* method, loop through the *items* collection and display each Game using the logger at *info* level.

_6. In the *com.example* package, create a Spring Batch configuration class in the same fashion as in the last two parts of lab. Then complete the class:

a. Autowire JobBuilderFactory and StepBuilderFactory fields.

b. Write an @Bean method that returns an instance of your custom ItemReader class.

c. Write an @Bean method that returns an instance of your custom ItemWriter class.

d. Write an @Bean method that returns a configured Step that uses a chunk size of three and your ItemReader and ItemWriter objects.

e. Write an @Bean method that returns a configured Job in the same fashion as the last lab.

_7. In your command prompt or Terminal, use *bootRun* to run the batch job. You should see output for three games.

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 5: Introduction to Spring Integration

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

Objectives:

- To use Spring Integration

Part 1: A Hello, World Spring Integration Application

Steps:

_1. Open a command-prompt or Terminal and change to **{Lab Installation Directory}/lab05**.

_2. Use the Spring Boot CLI to generate a starter project with these specifications:

```
Dependencies:       integration
Build language:     Gradle
Options:            Expanded project
```

_3. Edit the *build.gradle* and modify it:

a. In the *jar* task configuration, change the *baseName* to **lab05**.

b. Delete the *eclipse{...}* block at the end.

_4. Generate an Eclipse project and import it into Eclipse.

_5. Create the rudimentary service layer for this application:

a. In the *hello* package, create an interface named **HelloWorld** that contains a single method:

```
void sayHello(String name);
```

b. In the same package, create a class named **HelloWorldImpl** that implements the interface by printing "Hello, xxxx", where "xxxx" is the name to the console.

_6. In the *com.example* package, create a class named **IntegrationConfiguration**, then complete it:

a. Annotate the class:

```
@Configuration
@EnableIntegration
```

The @Configuration annotation is from core Spring and indicates this class provides Java configuration.

The @EnableIntegration annotation enables Spring Integration so you can define artifacts such as Message Channnels using Java configuration.

b. Write a configuration method that returns an instance of your service-layer class:

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
@Bean
public HelloWorld hello()
{
    return new HelloWorldImpl();
}
```

c. Write a configuration method that returns an instance of Spring Integration's *DirectChannel*, which is a simple *Point-to-Point* channel:

```
@Bean
public MessageChannel shippingChannel()
{
    return new DirectChannel();
}
```

d. Write a configuration method that returns an instance of Spring Integration's MessageHandler. Also configure this bean as a *Service Activator* configured to talk to your Hello, World service:

```
@Bean
@ServiceActivator(inputChannel="shippingChannel")
public MessageHandler activator()
{
    MethodInvokingMessageHandler mh =
      new MethodInvokingMessageHandler(hello(), "sayHello");

    return mh;
}
```

_7. Modify your program's *main()* method so it sends a message to the channel you configured, thus invoking your Hello, World service:

a. In DemoApplication.java, in the *main()* method, replace the generated code so Spring Boot omits the banner and to retrieve a Spring application context reference:

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

b. Use the application context to retrieve a reference to the channel:

```
MessageChannel channel =
  ctx.getBean("shippingChannel", MessageChannel.class);
```

c. Create a simple messaging containing your name:

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
Message<String> message =
   MessageBuilder.withPayload("xxxxxxx").build();
```

Substitute your name for the xxxxxx.

d. Send the message to the channel:

```
channel.send(message);
```

_8. To reduce the amount of informational messages (so you can see your service's output easier), edit the application.properties file and enter:

```
logging.level.org.springframework: WARN
logging.level.com.example: WARN
```

_9. Test your integration program. In your command prompt or Terminal, use *bootRun* to run - you should see your service's output.

Part 2: Using a Messaging Gateway

In the last part, the calling program (DemoApplication) had to be aware of messaging and Spring Integration - it had to create a Spring Integration message and send it.

In this part, you will configure a *Messaging Gateway* that encapsulates your Hello, World service, so the calling program can use the service's native API instead of explicitly sending messages. You will also modify the service so it returns the String instead of directly printing it to the console.

Steps:

_1. First, edit the HelloWorld interface and change the method signature to:

```
public String sayHello(String name);
```

Then modify the HelloWorldImpl class so it returns a String instead of writing it to the console.

_2. Next, edit the the IntegrationConfiguration classes and make the following changes:

a. Above the class definition, write another annotation that tells Spring Integration to find classes and interfaces that act as integration components:

```
@IntegrationComponentScan("hello")
```

b. In the *activator* method, comment out the code that creates the message handler, and replace it with:

```
ServiceActivatingHandler mh =
  new ServiceActivatingHandler(hello(), "sayHello");
```


Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

We need to do this since the MethodInvokingMessageHandler only supports messages that return *void* - i.e. unidirectional messages.

Note that we configure this message handler to reference the service implementation class.

_3. Next, edit HellowWorld.java (the interface) and above the interface definition, write this annotation:

```
@MessagingGateway(defaultRequestChannel="shippingChannel")
```

This configures the interface as a Spring Integration Gateway. At runtime, Spring Integration will create a proxy class instance from the interface, then call your actual service method.

_4. Finally, in DemoApplication.java, comment out the three lines that create and send a message, and replace them with:

```
HelloWorld hw = ctx.getBean("hello", HelloWorld.class);
String s = hw.sayHello("Sam");
System.out.println(s);
```

Now how we retrieve a bean using the interface type, HellowWorld. Spring Integration actually returns a proxy that invokes your service activator.

The code then calls sayHello() on the proxy, at which point Spring Integration creates a message, routes it to the service activator's configured channel (shippingChannel).

You have configured a ServiceActivatingHandler on the channel - it receives the message, unpackages it, then calls the sayHello() method on your service implementation class.

So the end result is the same, but now the calling program doesn't have to know anything about messaging, since the gateway encapsulates Spring Integration (the messaging system) using the service's native API.

_5. Run the program as before. It should have the same output.

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 6: Spring Integration Messages and Channels

In this lab, you will write Spring Integration applications that use different types of channels.

Objectives:

- To use Spring Integration channels

Part 1: Using Point-to-Point Channels

Steps:

_1. Open a command-prompt or Terminal and change to **{Lab Installation Directory}/lab06**.

_2. Use the Spring Boot CLI to generate a starter project with these specifications:

```
Dependencies:       integration
Build language:     Gradle
Options:            Expanded project
```

_3. Edit the *build.gradle* and modify it:

a. In the *jar* task configuration, change the *baseName* to **lab06**.

b. Delete the *eclipse{...}* block at the end.

_4. Generate an Eclipse project and import it into Eclipse.

_5. Create the domain layer for the application:

a. In the *com.example.domain* package, create a class named **Account** that implements java.io.Serializable.

b. Add fields for an account's data:

```
Field Name         Type
----------         ----

accountNumber      int
creationDate       java.util.Date
balance            java.math.BigDecimal
customerNumber     int
```

c. Generate get/set methods and a toString() method.

_6. Create the service layer for this application:

a. In the *com.example.service* package, create an interface named **AccountService** that contains a single method:

```
void accrueInterest(Account a);
```


Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

b. In the *com.example.service* package, create a class named **AccountServiceImpl** that implements the interface.

The implementation should print the account to the console, increase the balance by 5%, then print the modified account object to the console. To increase the balance, you can use code like:

```
BigDecimal balance = a.getBalance();
balance = balance.add(balance.multiply(new BigDecimal(0.05)));

a.setBalance(balance);
```

_7. In the *com.example* package, create a class named **IntegrationConfiguration**, then complete it:

a. Annotate the class with @Configuration and @EnableIntegration.

b. As you did in the last lab, write a configuration method that returns an instance of your service-layer class.

c. As you did in the last lab, write a configuration method that returns an instance of Spring Integration's *DirectChannel*, which is a simple *Point-to-Point* channel.

d. As you did in the last lab, write a configuration method that returns an instance of Spring Integration's MessageHandler. Also configure this bean as a *Service Activator* configured to talk to your ProcessAccount service's *accrueInterest()* method.

_8. Modify your program's *main()* method so it sends a message to the channel you configured, thus invoking your Hello, World service:

a. As in the last lab, in DemoApplication.java, in the *main()* method, replace the generated code so Spring Boot omits the banner and to retrieve a Spring application context reference.

b. Use the application context to retrieve a reference to the channel you configured earlier.

c. Create an Account object and initialize the fields to values of your choice.

d. Create a simple messaging containing the account object. Assuming your Account object's reference name is *a*:

```
Message<Account> message =
            MessageBuilder.withPayload(a).build();
```

e. Send the message to the channel.

_9. To reduce the amount of informational messages (so you can see your service's output easier), edit the application.properties file and enter:

```
logging.level.org.springframework: WARN
logging.level.com.example: WARN
```

_10. Test your integration program. In your command prompt or Terminal, use *bootRun* to run - you should see your service's output.

_11. Now let's add a header to the message.

a. First, modify the AccountService interface so that the *accrueInterest()* method accepts an additional parameter of type Long named **expiry**.

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

b. Then modify the AccountServiceImpl class so that the *accrueInterest* method signature specifies the new parameter, annotating it so that Spring Integration extracts the header's value for you:

```
public void accrueInterestAccount(Account a,
   @Header(name="expirationDate") Long expiry)
```

Note: You will need to import *org.springframework.messaging.handler.annotation.Header*.

Modify the method so that it also prints the message expiration to the console. Note that Spring Integration passes dates as Long integers - you can create a java.util.Date from the Long if you want to display the date instead of the integer.

c. In DemoApplication, before sending the message, create an object representing a week from the current date:

```
Calendar now = Calendar.getInstance();
now.add(Calendar.DATE, 7);
```

Then modify the MessageBuilder so it sets this as the message's *expirationDate* header:

```
Message<Account> message =
           MessageBuilder.withPayload(a)
           .setExpirationDate(now.getTime())
           .build();
```

d. Run the application - you should see the expiration date header's value.

_12. The application worked OK, but everything is running synchronously on the same thread since you used a DirectChannel. If the sender needed to send many messages quickly, it would have to wait for each to be delivered if the service was slow. To demonstrate:

a. Edit AccountServiceImpl and after setting the new balance, but before printing the modified message, introduce an artificial delay:

```
TimeUnit.SECONDS.sleep(10);
```

b. Modify DemoApplication so it sends two messages:

```
System.out.println("Sending first message");
channel.send(message);

System.out.println("Sending second message");
channel.send(message);
```

c. Run the application and note that the sender must wait for the first message to be processed before it can send the second.

_13. Now let's convert to using a QueueChannel instead of a DirectChannel. QueueChannels queue messages if there's no recipient available. Follow these steps:

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

a. In IntegrationConfiguration, in the method that creates the DirectChannel, comment out the channel creation and instead create and return a QueueChannel using the one-argument constructor, passing the value 10. This creates a QueueChannel with a capacity of 10 buffered messages.

b. Try running the application - what happens? You should get an error indicating that there's no "poller" defined. To define the poller, in IntegrationConfiguration, define a **default** poller:

```
@Bean(name = PollerMetadata.DEFAULT_POLLER)
public PollerMetadata defaultPoller()
{

    PollerMetadata pollerMetadata = new PollerMetadata();
    pollerMetadata.setTrigger(new PeriodicTrigger(100));
    return pollerMetadata;
}
```

This creates a default poller for the application that will check the queue every 100 milliseconds.

c. Run the application again - you should see the sender sending both messages asynchronously and then message-processing kick in after the artificial delays.

Note that since your application is now using threads, the JVM will stay alive even after the main() method exits. You can kill the program by pressing Ctrl+C in the command prompt and then pressing the 'Y' key.

You can remove the artificial delay from the service implementation before continuing.

Part 2: Adding a Publish-Subscribe Channel

In the last part, you coded an AccountService messaging system that used a Point-to-Point channel. In this part, you will add a Publish-Subscribe channel to the application.

The scenario is that after accruing interest, the application needs to notify TWO interested parties: an *audit* service and a service that emails the customer. Neither of these will actually do much at this point, but you will gain experience configuring Spring Integration. Figure 1 shows the messaging system you will implement:

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

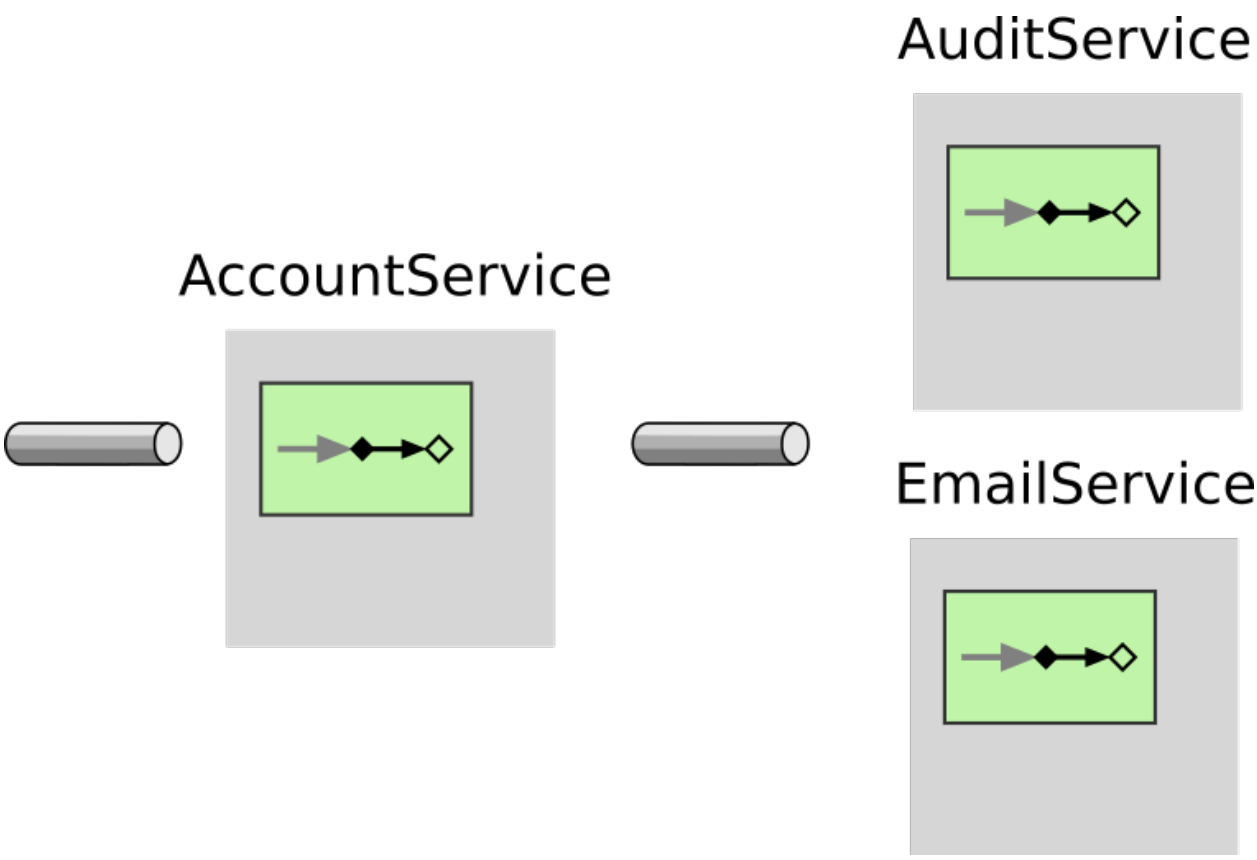


Figure 1: Part 2 Messaging System

Steps:

_1. First, modify the AccountService interface and implementation so that the accrueInterest() method returns the modified Account instead of void. We do this so Spring Integration can write the modified Account object to the new output channel you will create.

_2. In the *com.example.service* package, define an interface and implementation for the *audit* service. The service should have a single method that accepts an Account, returning *void*. The method should verify that the balance is greater than zero and display a message to the console indicating whether the audit passed or not.

To check the balance, you can use code like:

```
if(balance.compareTo(new BigDecimal(0.0)) == -1)
{
  // balance less than zero
  . . .
}
```

_3. In the *com.example.service* package, define an interface and implementation for the *email* service. The service should have a single method that accepts an Account returning void. The method shoud simulate sending an email by simply printing out the account's customerNumber.

_4. Modify IntegrationConfiguration:

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

a. Write an @Bean configuration method that creates another channel named **changedAccountChannel**. The method should return an instance of *PublishSubscribeChannel*.

b. Modify your existing ServiceActivator so it calls accrueInterest() and sends the modified message to the channel you just configured:

```
@Bean
@ServiceActivator(inputChannel="accountChannel")
public MessageHandler activator()
{
//     MethodInvokingMessageHandler mh =
//       new MethodInvokingMessageHandler(accountService(), "accrueInterest");

  ServiceActivatingHandler mh =
      new ServiceActivatingHandler(accountService(), "accrueInterest");

  mh.setOutputChannel(changedAccountChannel());
  return mh;
}
```

c. Write an @Bean configuration method that creates an instance of your audit service.

d. Write an @Bean configuration method that configures a ServiceActivator for the audit service. Be sure to configure changedAccountChannel as the input channel.

e. Repeat the above two steps for the *email* service.

_5. Run your program - you should see the output from your audit and email services.

_6. Modify DemoApplication so it sends a message that fails audit. Create a new Account object with a balance less than zero and assemble it into a new message.

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 7: Spring Integration Endpoints

In this lab, you will write a Spring Integration application that use different types of endpoints.

The application you will implement has the following flow:

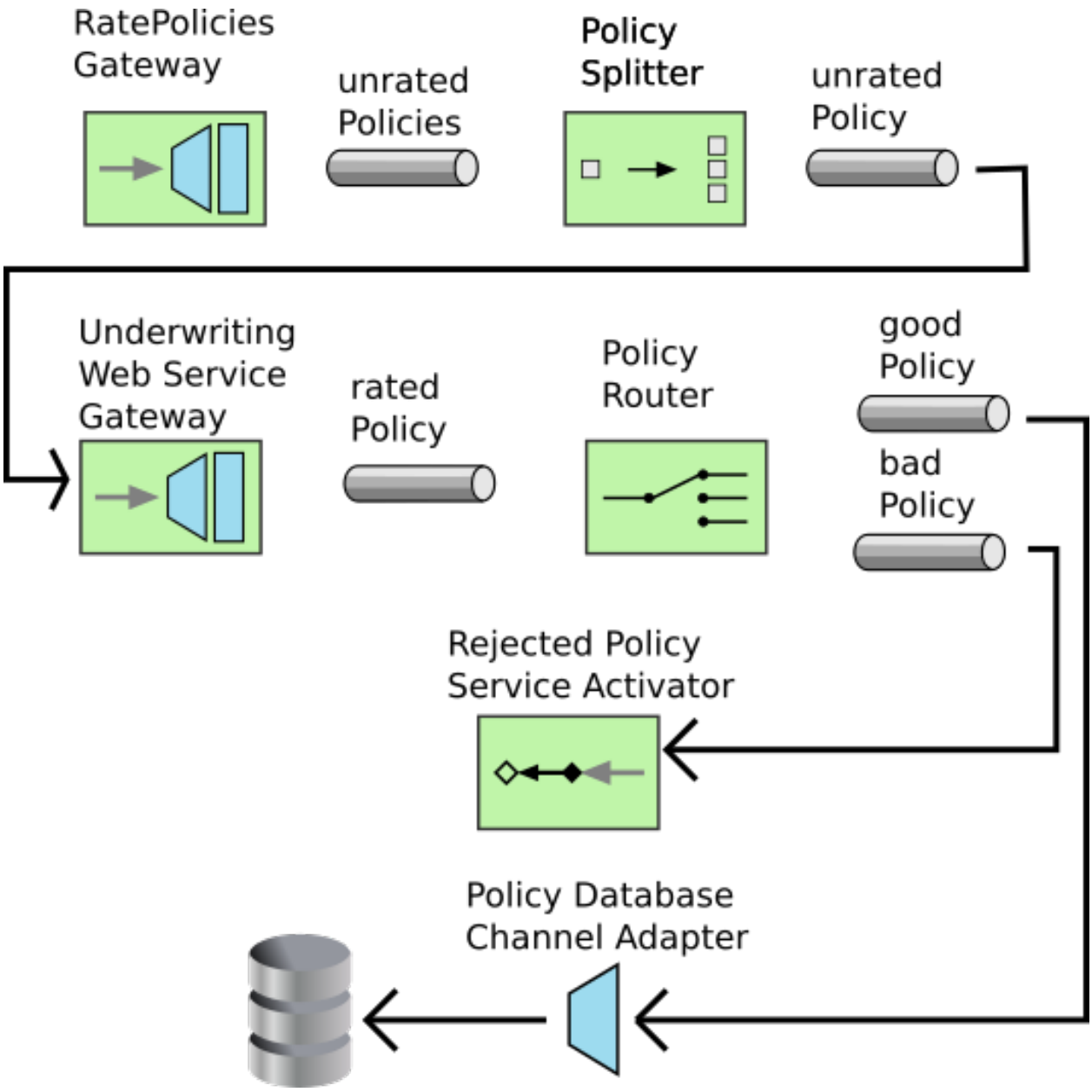


Figure 1: Application Flow

The flow starts when something calls the RatePolicies gateway passing a list of insurance policies. In your lab, you will write a simple main() method as the client, but in the real world, it might be a Web application that lets an insurance agent batch up a bunch of new policies for processing. The gateway puts the policy list onto the unratedPolicies channel.

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

You will write a PolicySplitter that splits the list into individual policies, writing each to the unratedPolicy channel. Next, you will configure a Web Service gateway that subscribes to the channel, invokes an existing SOAP-based Web service that underwrites (rates) the policy, returning an updated policy containing the rating results. The gateway then sends the modified policy to the ratedPolicy channel.

You will write a PolicyRouter that acts as a *Content Router*, sending "accepted" policies to the the goodPolicy channel and rejected policies to the badPolicy channel.

You will write a RejectedPolicy *Service Activator* that displays the underwriting result to the console. In the real world, we would do additional work, e.g. send an email to agent.

You will configure a JDBC *Outbound Channel Adapter* to write accepted policies to a database table.

Objectives:

- To use Spring Integration endpoints, including routers, gateways and splitters

Part 1: Project and Database Setup

Steps:

_1. You must configure the database for your program. We will use the free, open-source HypersonicSQL database to hold bank account information. Follow these steps to configure the database:

Windows Instructions:

a. Open a command-prompt window (choose Start - Run and type **cmd**), then change to the **{Lab Installation Directory}/lib** directory. If your computer has the standard lab setup:

```
c:
cd \Users\username\advspring\lib
```

Substitute your actual user name, e.g. *libtrain1* for the "username" above.

b. Start the HypersonicSQL server running:

```
java -cp hsqldb-2.3.3.jar org.hsqldb.Server -database lab07db
```

The server will start and display a sign-on message. If prompted by Windows, allow access for the server.

Note: Be sure to leave this command-prompt running until you are finished with the lab.

c. Open another command-prompt window and change to same directory as you did in the last step.

d. Start the HypersonicSQL administrator's GUI:

```
java -cp hsqldb-2.3.3.jar org.hsqldb.util.DatabaseManager
```

e. In the GUI, change the *Type* to **HSQL Database Engine Server**, then press OK. **Warning:** Be sure to choose the Server option!

f. In the GUI, in the command area between the Clear and Execute buttons, create and populate the database table, by entering the following (copy and paste from the 'starters' file):

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
CREATE TABLE policy(policyNumber INT IDENTITY PRIMARY KEY,
   amount DOUBLE, customerNumber INT, status VARCHAR(100));
```

Then press the Execute button.

g. From the menu, choose *View - Refresh Tree* to see the table you just created.

Mac OS X Instructions:

a. In the Finder, open your /Applications/Utilities folder, then start Terminal. In the Terminal, change to the **{Lab Installation Directory}/lib** directory. If your computer has the standard lab setup:

```
cd /Users/username/advspring/lib
```

Substitute your actual user name, e.g. *libtrain1* for the "username" above.

b. Start the HypersonicSQL server running:

```
java -cp hsqldb-2.3.3.jar org.hsqldb.Server -database lab07db
```

The server will start and display a sign-on message.

Note: Be sure to leave this Terminal running until you are finished with the lab.

c. Open another Terminal and change to same directory as you did in the last step.

d. Start the HypersonicSQL administrator's GUI:

```
java -cp hsqldb-2.3.3.jar org.hsqldb.util.DatabaseManager
```

e. In the GUI, change the *Type* to **HSQL Database Engine Server**, then press OK. **Warning:** Be sure to choose the Server option!

f. In the GUI, in the command area between the Clear and Execute buttons, create and populate the database table, by entering the following (copy and paste from the 'starters' file):

```
CREATE TABLE policy(policyNumber INT IDENTITY PRIMARY KEY,
   amount DOUBLE, customerNumber INT, status VARCHAR(100));
```

Then press the Execute button.

g. From the menu, choose *View - Refresh Tree* to see the tables you just created.

_2. Next, import and start the Rate Policy SOAP-based Web service:

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

b. On the first wizard page, choose *General - Existing Projects into Workspace*, then press Next.

c. On the next page, click the radio button for *Select archive file*, then browse to **{Lab Installation Directory}/starters/lab07** and select *PolicyWebService.zip*. Press Open followed by Finish.

You should see the PolicyWebService project in the Package Explorer.

d. Examine the imported project and see if you can make sense of it. This is a JAX-WS SOAP Web service. The actual service implementation is in the PolicyService class. Note how it exposes the ratePolicy()

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

method as service method and that method uses the coverage amount to either accept or reject the input policy.

e. To start the Web service, right-click on PolicyPublisher.java and choose *Run As - Java Application*. The service should start and display a couple diagnostic messages in the Eclipse console.

Leave the service running for the rest of the lab.

f. To verify that the Web service is functional, open a Web browser and enter this URL:

```
http://localhost:8085/ratePolicy?WSDL
```

You should see the Web Services Description Language XML that documents the Web service.

_3. Open a command-prompt or Terminal and change to **{Lab Installation Directory}/lab07**.

_4. Use the Spring Boot CLI to generate a starter project with these specifications:

```
Dependencies:       integration
Build language:     Gradle
Options:            Expanded project
```

_5. Edit the *build.gradle* and modify it:

a. In the *jar* task configuration, change the *baseName* to **lab07**.

b. Add the following to the *dependencies* block:

```
runtime('org.hsqldb:hsqldb')
compile('org.springframework.integration:spring-integration-ws');
compile('org.springframework.integration:spring-integration-jdbc');
compile ('commons-dbcp:commons-dbcp:1.4')
```

c. Delete the *eclipse{...}* block at the end.

_6. Generate an Eclipse project and import it into Eclipse in the same way you have done in earlier labs.

Note: When importing, specifying the root directory, you may need to uncheck the *Copy projects into workspace* setting, then press Refresh so you can import the *lab07* project.

_7. To reduce the amount of informational messages (so you can see your program's output easier), edit the application.properties file and enter:

```
logging.level.org.springframework: WARN
logging.level.com.example: WARN
```

Part 2: Building the Application

Steps:

_1. First, create the domain layer, complete with necessary artifacts so objects can work as a parameters to the Web service:

a. To save time, you can copy the InsurancePolicy class from the imported project.

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

First create a package named **com.example.domain** in the lab07 project, then copy and paste InsurancePolicy.java from the PolicyWebService project into that package.

Study the class and ensure you understand it. Note that it's annotated with @XmlRootElement - that enables it to work the JAX-B Web service infrastructure.

b. In the *com.example.domain* package, create a new class named **ObjectFactory**. Note that it's important to use this exact name so the JAX-B infrastructure can find it.

Annotate the class with @XmlRegistry.

Write the following method that JAX-B will use to create objects:

```
public InsurancePolicy createInsurancePolicy()
{
    return new InsurancePolicy();
}
```

c. Right-click on the *com.example.domain* package and create a new file named **package-info.java**. Be sure to use that exact name.

Add the following to the file:

```
@javax.xml.bind.annotation.XmlSchema(
  namespace = "http://goliath.com/",
  elementFormDefault = javax.xml.bind.annotation.XmlNsForm.QUALIFIED)
package com.example.domain;
```

This allows the Spring Web service gateway to correctly create the XML SOAP message to invoke the Web service.

_2. Start the configuration for the application. As usual, we will use Java-based configuration instead of XML:

a. In the *com.example* package, create a class named **IntegrationConfiguration**.

b. Annotate the class:

```
@Configuration
@EnableIntegration
@IntegrationComponentScan("com.example.service")
```

_3. Next, create the RatePolicies *Message Gateway* that encapsulates the application so clients don't need to understand messaging. This is similar to what you did in the first Spring Integration lab for the Hello service:

a. In the *com.example.service* package, create a new interface named **RatePolicies**.

b. Annotate the interface with @MessagingGateway.

c. Write a method definition that clients can call (Spring Integration will generate a proxy that implements the gateway by sending a message to a channel):

```
void rate(Collection<InsurancePolicy> policies);
```

Annotate the method to specify the channel:

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
@Gateway(requestChannel="unratedPoliciesChannel")
```

d. As you did in the last lab, in IntegrationConfiguration.java, define a bean-creation method that creates a DirectChannel named *unratedPoliciesChannel*.

_4. Next, code the client that will invoke the gateway:

a. Edit DemoApplication.java, and replace the existing code in main() with:

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

b. Retrieve the proxied gateway from application context:

```
RatePolicies rater = (RatePolicies)ctx.getBean(RatePolicies.class);
```

c. Write code that creates an ArrayList of InsurancePolicy objects. Your code should create and initialize at least two InsurancePolicy objects: one with coverageAmount of 1,000, the other with 100,000. Set the customerNumber to anything you want, and the status to Status.Unrated. You don't need to assign a policyNumber since that will later be assigned by the database.

Be sure to add each policy object to the collection, then call the rate() method on the RatePolicies reference, passing the collection.

d. You can try running your application at this point - it should fail indicating that the unratedPoliciesChannel has no subscribers.

_5. Next, code and configure the PolicySplitter that accepts the list and sends individual policies to the unratedPolicyChannel:

a. In the *com.example.service* package, create a new class named **PolicySplitter**.

b. Annotate the class with @MessageEndpoint.

c. Write the method that splits the list (in this case, it doesn't really need to do much, since Spring Integration knows how to iterate collections and pass the individual elements to the output channel):

```
public List<InsurancePolicy> split(List<InsurancePolicy> policies)
{
    return policies;
}
```

d. Annotate the method:

```
@Splitter(inputChannel = "unratedPoliciesChannel",
  outputChannel = "unratedPolicyChannel")
```

e. In IntegrationConfiguration.java, create another DirectChannel, this one named **unratedPolicyChannel**.

f. Again, you can try running the app - it should fail indicating that unratedPolicyChannel has no subscribers.

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

_6. Next, configure the Web services gateway to invoke the RatePolicy Web service:

a. In IntegrationConfiguration.java, write a bean-creation method that knows how to use the JAX-B runtime to convert an InsurancePolicy object into XML:

```
@Bean
public Jaxb2Marshaller jaxb2Marshaller()
{
    Jaxb2Marshaller marshaller = new Jaxb2Marshaller();
    marshaller.setContextPath("com.example.domain");
    return marshaller;
}
```

b. Still in IntegrationConfiguration.java, write a bean-creation method that for a Service Activator that will act as the gateway:

```
@Bean
@ServiceActivator(inputChannel = "unratedPolicyChannel")
public MessageHandler wsOutboundGateway()
{

}
```

You will have a compile error because you haven't yet returned anything.

c. In the wsOutboundGateway() method, create the gateway object itself, connecting it to the Web service:

```
MarshallingWebServiceOutboundGateway gw =
   new MarshallingWebServiceOutboundGateway(
        "http://localhost:8085/ratePolicy", jaxb2Marshaller());
```

d. Still in the wsOutboundGateway() method, configure the gateway's output channel and return it:

```
gw.setOutputChannelName("ratedPolicyChannel");

return gw;
```

e. Still in IntegrationConfiguration.java (but NOT in the wsOutboundGateway() method), create yet another DirectChannel, this one named **ratedPolicyChannel**.

f. You can try running your incomplete application at this point - it should fail indicating that ratedPolicyChannel has no subscribers.

You should also be able to look in the Eclipse console and see that the Web service received a request.

_7. Next, create the PolicyRouter that acts as a **Content Router**. The router will send policies that the Web service approved to one channel, and rejected policies to another:

a. In the *com.example.service* package, create a class named **PolicyRouter**.

b. Annotate the class with @MessageEndpoint.

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

c. In the class write a method that accepts an InsurancePolicy object and returns a String. The method should return *goodPolicyChannel* if the input policy has the Approved status and return *badPolicyChannel* otherwise.

   Spring Integration will examine the returned String and resend the policy to the appropriate channel.

d. Annotate the method so it's a Router and to specify the input channel:

```
@Router(inputChannel="ratedPolicyChannel")
```

e. In IntegrationConfiguration.java, define two more DirectChannels: one named **goodPolicyChannel**, the other **badPolicyChannel**.

f. You can try running the app at this point - it should fail indicating that the new channels have no subscribers.

_8. Next, code the service and Service Activator for rejected policies. In a real-world application, this service would take some action, perhaps emailing the agent, but yours will just write the rejected policy's info to the console:

a. In the *com.example.service* package, create an interface named **RejectedPolicyService**.

   In the interface, define a single method that accepts an InsurancePolicy and returns void.

b. In the same package, create a class named **RejectedPolicyServiceImpl** that implements the interface.

   In the method body, print out the customerNumber and the ratingComment.

c. In IntegrationConfiguration, write a bean-creation method that returns an instance of the RejectedPolicyServiceImpl class. The method's return type should be the interface: RejectedPolicyService.

d. Still in IntegrationConfiguration, write a bean-creation method that configures a Service Activator using a MethodInvokingMessageHandler. Do this in the same fashion as in the last lab. The handler should invoke the method in the RejectedPolicyService interface you just wrote.

   Be sure to annotate with @Bean and @ServiceActivator, specifying **badPolicyChannel** as the input channel.

_9. Next, configure the JDBC *Outbound Channel Adapter* that will write approved policy data to the database table you created earlier:

a. In IntegrationConfiguration.java, write a bean-creation method that defines a DataSource for the HSQL database. Do this in the same fashion as you did in Lab 4, Part 2.

b. Next, write the skeleton of a bean-creation method that acts as a ServiceActivator for the outbound channel adapter:

```
@Bean
@ServiceActivator(inputChannel = "goodPolicyChannel")
public MessageHandler approvedPolicyActivator()
{

}
```

   You will have compile errors since you haven't yet returned anything.

c. In the method, first create the JDBC outbound channel adapter:

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
JdbcMessageHandler jh = new JdbcMessageHandler(dataSource(),
        "INSERT INTO policy (amount,customerNumber,status)"
                + " VALUES(?, ?, ?)");
```

Note how we write the SQL to insert a policy's data into the table you created earlier. We don't provide the policyNumber, since that's an auto-generated value in the database.

Spring JDBC uses JDBC PreparedStatements - we will need to supply the values for the three question-mark placeholders.

d. Still in the method, start writing a local inner class that will provide the values for the question-mark placeholders:

```
class MyPreparedStatementSetter
   implements MessagePreparedStatementSetter
{

}
```

After importing MessagePreparedStatementSetter, place the text-entry cursor within the inner class, then from the Eclipse menu, choose *Source - Override/Implement methods*, then press OK. Eclipse writes an empty setValues() method.

In setValues(), first retrieve the InsurancePolicy from the message payload:

```
InsurancePolicy ip = (InsurancePolicy) requestMessage
        .getPayload();
```

Then provide the placeholder values in the PreparedStatement:

```
ps.setDouble(1, ip.getCoverageAmount());
ps.setInt(2, ip.getCustomerNumber());
ps.setString(3, ip.getStatus().name());
```

e. After the inner-class closing brace, but still within the approvedPolicyActivator() method, create an instance of the inner class and assign it to the message handler. Then return the message-handler instance:

```
jh.setPreparedStatementSetter(
   new MyPreparedStatementSetter());

return jh;
```

_10. Run the application - you should see a message for any rejected policies (any with coverageAmount = 100,000).

Go back to the HSQLDB admin GUI and execute *SELECT * FROM policy*. You should see the data for for approved policies (e.g. one with coverageAmount = 1,000).

You can kill the Web service in Eclipse when you are done - press the red-square icon in the console pane's toolbar.

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 8: Introduction to Spring XD

In this lab, you will get started with Spring XD.

Objectives:

- To start Spring XD in single-node mode
- To create an execute a simple stream

Part 1: Getting Started with Spring XD

Steps:

_1. Ask your instructor for the following information:

Lab Installation directory: _____

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

```
Spring XD Installation directory:  c:\Users\username\java\spring-xd-1.3.0.RELEASE
```

Substitute your actual username for the placeholder "username".

If your computer is running Mac OS X, the directories will be something like:

```
Spring XD Installation directory:  /Users/username/java/spring-xd-1.3.0.RELEASE
```

Substitute your actual username for the placeholder "username".

_2. Open a command-prompt or Terminal and change to **{Lab Installation Directory}/xd/bin**.

Start Spring XD in single-node mode:

Windows Instructions:

```
xd-singlenode.bat
```

Mac OS X Instructions:

```
./xd-singlenode
```

Spring XD will start and display a banner. Leave this window open for the remainder of the lab.

_3. Open a new command-prompt or Terminal and change to **{Lab Installation Directory}/shell/bin**.

Start the XD command-line running:

Windows Instructions:

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
xd-shell.bat
```

Mac OS X Instructions:

```
./xd-shell
```

The command line will start and display a banner followed by an *xd:>* prompt.

Note: If you see a prompt *server-unknown:>*, type **exit** and then ensure that the single-node server has finished starting up. Then start the command-line again.

_4. In the command line, create a simple stream:

```
stream create --definition "time | log" --name ticktock --deploy
```

This creates a stream that uses the *time* module as the input and the *log* module as the output. These modules are predefined in Spring XD: *time* generates a message every second and *log* usaes a logging framework to log a message at INFO level.

We name the stream **ticktock** and tell Spring XD to deploy it immediately.

Look in the original command-prompt window or Terminal where you started Spring XD itself - you should see the logged output of the stream.

_5. Start the Spring XD Web GUID by opening a Web browser and navigating to:

```
http://localhost:9393/admin-ui
```

Press the *STREAMS* link at the top of the page. You should see your ticktock stream display. Press the *Destroy* button to get rid of the stream.

Look back in the XD window - the periodic log messages have ceased.

_6. Next, let's experiment with some commands in the command line window:

a. Display information about the admin server:

```
admin config info
```

b. Display info about the XD single-node server:

```
runtime containers
```

c. Display the (empty) list of streams:

```
stream list
```

d. List all of the installed *modules*:

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
module list
```

Find the *time* and *log* modules in the list.

e. Display info about the *time* module:

```
module info  source:time
```

What is the default period of the time module? How would you change the default?

f. Repeat the above for the *log* sink module.

What's the default logging level? How would you change the default?

_7. Using the info you just learned, recreate the *ticktock* stream, but with messages every five seconds at WARN level.

Hint: To specify a configuration option for a module, prepend the option name with '--' and specify the value after an equals sign, for example:

```
somemodule --someoption=somevalue
```

_8. Use the *stream all destroy* command to destroy the stream.

_9. In the command window, enter **exit** to terminate the command shell. Then press Ctrl+C on the window where Spring XD is running.

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Lab 9: Spring XD Modules

In this lab, you use the Spring XD command shell to create streams using predefined Spring XD modules.

Objectives:

- To use Spring XD modules

Part 1: Creating a Stream with Modules

Steps:

_1. The stream you will create will read from a database, so start by creating the input data:

a. Open a command prompt or Terminal, then change to **{Lab Installation Directory}/lib**.

b. Start the HSQLDB database server running:

```
java -cp hsqldb-2.3.3.jar org.hsqldb.Server -database lab09db
   -port 9002
```

NOTE: So it fits on the printed page, we have split the command across two lines. Please enter it on a single line. Also note that we specify 9002 for the server's port so it doesn't conflict with the HSQLDB server used internally by Spring XD.

c. In a new command-prompt or Terminal, change to the same directory, then start the HSQLDB admin GUI:

```
java -cp hsqldb-2.3.3.jar org.hsqldb.util.DatabaseManager
```

In the Connect dialog, enter or select:

```
Type:     HSQL Database Engine Server
Driver:   org.hsqldb.jdbcDriver
URL:      jdbc:hsqldb:hsql://localhost:9002/
User:     SA
Password: -- leave empty --
```

NOTE: Be sure to choose the correct type and also ensure that the URL has the trailing slash.

d. In the GUI, create a table:

```
CREATE TABLE account (accountNum INTEGER IDENTITY PRIMARY KEY,
createDate DATE, balance DOUBLE, open BOOLEAN,
holderName VARCHAR(100));
```

e. Insert some sample data:

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
INSERT INTO account (createDate, balance, open, holderName)
  VALUES ('2012-5-6', 300, true, 'Sam Smith');
INSERT INTO account (createDate, balance, open, holderName)
  VALUES ('2013-6-7', 200, false, 'Sue Jones');
INSERT INTO account (createDate, balance, open, holderName)
  VALUES ('2011-2-3', 1000, true, 'George Fish');
INSERT INTO account (createDate, balance, open, holderName)
  VALUES ('2010-4-5', 0, false, 'Noah Cash');
```

_2. As you did in the last lab, start Spring XD running in its single-node configuration, then start the XD command shell.

_3. Now let's do a quick test. Using the JDBC Source example in the book as a guide, in the Spring XD command shell, create and deploy a stream named **processAccounts** that uses the **jdbc** input source module and sends the output to the **log** output module. Here are the options you should specify for the **jdbc** module:

```
Option    Value
------    -----
url         jdbc:hsqldb:hsql://localhost:9002/
query       select * from account
username    sa
fixedDelay  100
```

Look in the command prompt or Terminal window where the Spring XD server is running. You should see lines that look something like:

```
2015-12-21T11:54:46-0500 1.3.0.RELEASE INFO task-scheduler-7 sink.processAccount
s - {ACCOUNTNUM=0, CREATEDATE=2012-05-06, BALANCE=300.0, HOLDERNAME=Sam Smith}
2015-12-21T11:54:46-0500 1.3.0.RELEASE INFO task-scheduler-7 sink.processAccount
s - {ACCOUNTNUM=1, CREATEDATE=2013-06-07, BALANCE=200.0, HOLDERNAME=Sue Jones}
2015-12-21T11:54:46-0500 1.3.0.RELEASE INFO task-scheduler-7 sink.processAccount
s - {ACCOUNTNUM=2, CREATEDATE=2011-02-03, BALANCE=1000.0, HOLDERNAME=George Fish
}
2015-12-21T09:26:00-0800 1.3.0.RELEASE INFO task-scheduler-7 sink.processAccount
s - {ACCOUNTNUM=3, CREATEDATE=2010-04-05, BALANCE=0.0, HOLDERNAME=Noah Cash}
```

This shows that the **log** module logged the four rows that the the **jdbc** module read from your database. Note that format of each message - the payload is a Map containing keys for each of the database column names with corresponding values from each table row.

Destroy the stream before continuing.

_4. Your next job is to add a **filter** to the stream that filters out rows for accounts with zero balance. Recreate the **processAccounts** stream, but configure a **filter** module after the **jdbc** module and before the **log** output module. Here are the options you should specify for the *filter* module:

```
Option      Value
------      -----
expression  'payload.balance > 0.00'
```


Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Note that since the payload is not JSON, we can just provide a Spring Expression that references the *balance* property on the payload object. Also note that you will need to enclose the expression in single quotes.

Look in the command-prompt or Terminal window - you should see log entries for all but the last row from the Accounts table.

Destroy the stream.

_5. Repeat the above step, but this time, filter out closed accounts.

Hint: Use the == operator in SpEL to check for equality.

Destroy the stream.

_6. Repeat the above, but this time use the **file** output module instead of the *log* module.

Then look in /tmp/xd/output for your log file. Note that this path is relative to the root of the volume where you started the XD server.

Destroy the stream.

_7. **Challenge:** In the previous streams, we set the fixedDelay to 100 seconds, which overrides the default poll rate of five seconds that the **jdbc** module uses to trigger database queries, which then read all of the rows.

In this part, you will instead configure the module so it uses an SQL update on a special column so that the module only reads new rows in the table.

a. Delete the processAccounts.log file in the tmp directory.

b. Drop and recreate the table so it has an additional BOOLEAN column named **newRow**.

Re-insert the four rows, initializing the new column to true for each.

c. Recreate the processAccount stream, but this time configure the query so it only includes rows with newRow set to true. Also configure the **update** option to set newRow to false. See the XD docs for the **jdbc** module for an example. You should also remove the fixedDelay option or set it to its default value of five seconds.

d. Examine the processAccounts.log file in the tmp directory - it should have two entries.

e. Use the HSQLDB admin GUI to insert another row into the Account table for an open account, specifying newRow = true.

f. Re-examine the log file - you should see the new entry. Then use the HSQLDB admin GUI to verify that all of the rows have newRow set to false.

Destroy the stream when you are done.

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Copyright © Descriptor Systems 2015. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.