

# **WA2747 Liberty Mutual Spring Microservices**

## **Student Labs**

### **Web Age Solutions Inc.**

## Table of Contents

Lab 1 - A Simple RESTful API in Spring Boot.....	3
Lab 2 - Use the Spring Web MVC Web Framework under Spring Boot.....	12
Lab 3 - Use the Spring JDBCTemplate under Spring Boot.....	18
Lab 4 - Use the Spring Data JPA under Spring Boot.....	24
Lab 5 - Learning the MongoDB Lab Environment.....	31
Lab 6 - Spring Data with MongoDB .....	36
Lab 7 - Create a RESTful API with Spring Boot.....	45
Lab 8 - Create a RESTful Client with Spring Boot.....	58
Lab 9 - Enable Basic Security.....	62
Lab 10 - Use AMQP Messaging with Spring Boot.....	66
Lab 11 - Use Netflix Eureka for Service Discovery.....	72
Lab 12 - Use Netflix Ribbon for Client-Side Load Balancing.....	82
Lab 13 - Use Netflix Hystrix for the Circuit Breaker Pattern.....	88
Lab 14 - EdgeComponents with Zuul .....	94
Lab 15 - Distributed tracing with Zipkin .....	105
Lab 16 - Spring Boot Project.....	120

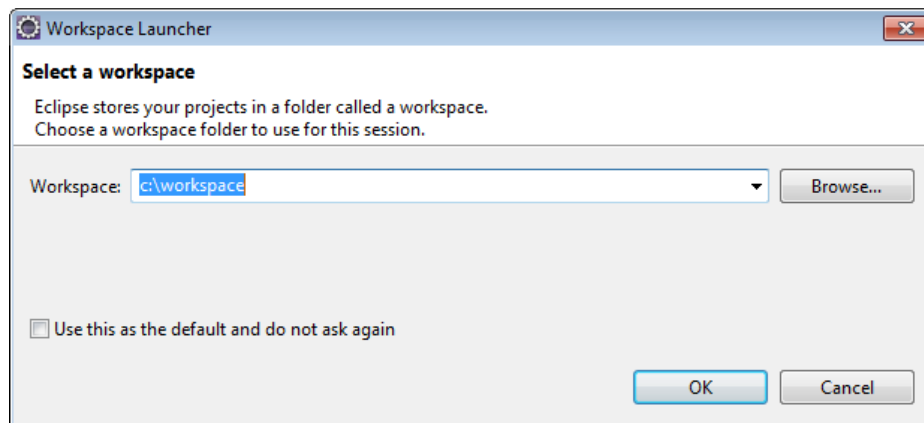
## Lab 1 - A Simple RESTful API in Spring Boot

In this lab we're going to build a simple "Hello World" API using Spring Framework and Spring Boot. The API will implement a single resource, "/hello-message" that returns a JSON object that contains a greeting.

### Part 1 - Create a Maven Project

We're going to start from scratch on this project, with an empty Apache Maven project, and add in the dependencies that will make a Spring Boot project with a core set of capabilities that we can use to implement our "Hello World" API.

- \_\_\_1. Open Eclipse by navigating to **C:\Software\eclipse** and double-clicking on **eclipse.exe** (note that the '.exe.' extension may not be shown, depending on the view options that have been set).
- \_\_\_2. In the **Workspace Launcher** dialog, enter 'C:\Workspace' in the **Workspace** field, and then click **OK**.

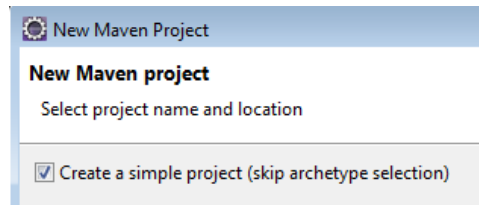


- \_\_\_3. Close the **Welcome** panel by clicking on the 'X':



- \_\_\_4. From the main menu, select **File** → **New** → **Maven Project**.

\_\_5. In the **New Maven Project** dialog, click on the checkbox to select "Create a simple project (skip archetype selection)", and then click **Next**.



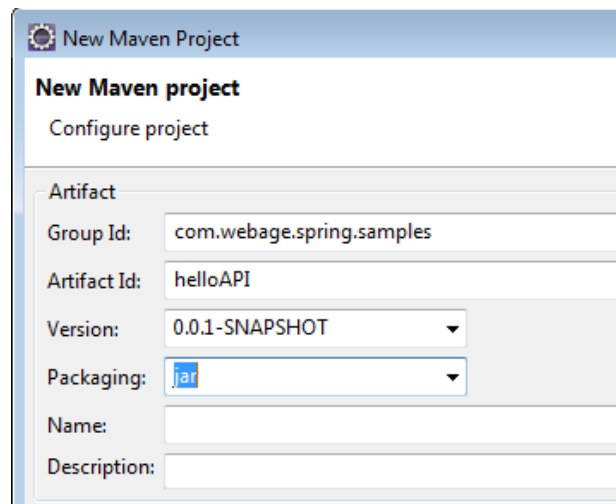
\_\_6. Enter the following fields:

**Group Id:** com.webage.spring.samples

**ArtifactId:** helloAPI

Leave all the other fields at their default values.

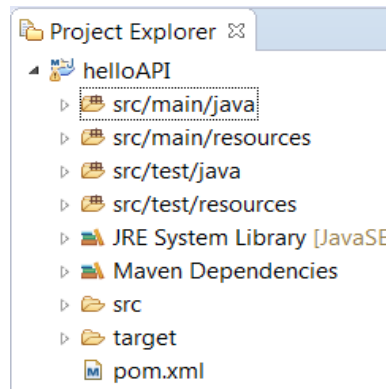
\_\_7. When the dialog looks like below, click **Finish**.



## Part 2 - Configure the Project as a Spring Boot Project

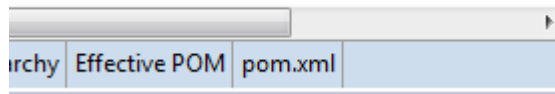
The steps so far have created a basic Maven project. Now we'll add the dependencies to make a Spring Boot project.

\_\_1. Expand the **helloAPI** project in the **Project Explorer**.



\_\_2. Double-click on **pom.xml** to open it.

\_\_3. At the bottom of the editor panel, click the **pom.xml** tab to view the XML source for **pom.xml**.



\_\_4. Insert the following text after the "<version>...</version>" element, and before the closing "</project>" tag.

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.4.2.RELEASE</version>
</parent>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

The entries above call out the Spring Boot Starter Parent project as the parent to this project, then call out the Spring Boot Starter Web dependencies. Finally the <build> element configures the Spring Boot Maven Plugin, which will build an executable jar file for the project.

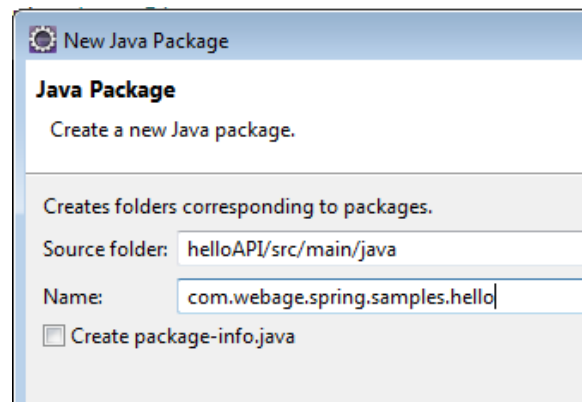
\_\_5. Save the file by pressing **Ctrl-S** or selecting **File -> Save** from the main menu.

### Part 3 - Create an Application Class

Spring Boot uses a 'Main' class to startup the application and hold the configuration for the application. In this section, we'll create the main class.

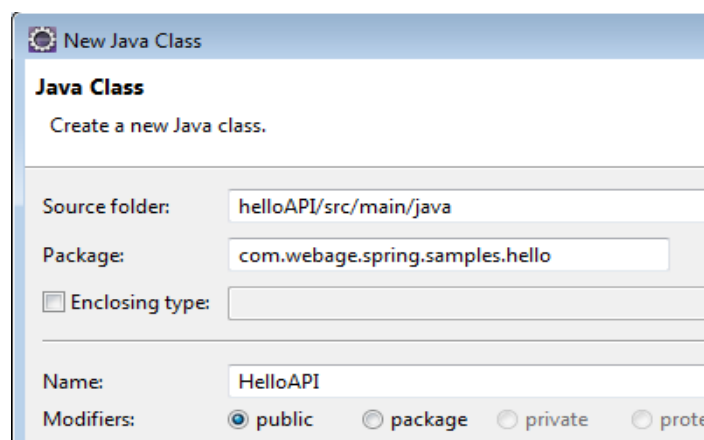
\_\_1. In the **Project Explorer**, right-click on **src/main/java** and then select **New -> Package**.

\_\_2. Enter 'com.webage.spring.samples.hello' in the **Name** field, and then click **Finish**.



\_\_3. In the **Project Explorer**, right-click on the newly-created package and then select **New -> Class**.

\_\_4. In the **New Java Class** dialog, enter 'HelloAPI' as the **Name**, and then click **Finish**.



\_\_5. Add the '@SpringBootApplication' annotation to the class, so it appears like:

```
@SpringBootApplication
public class HelloAPI {
```

\_\_6. Add the following 'main' method inside the class:

```
public static void main(String[] args) {
    SpringApplication.run(HelloAPI.class, args);
}
```

\_\_7. The editor is probably showing errors due to missing 'import' statements. Press **Ctrl-Shift-O** to organize the imports.

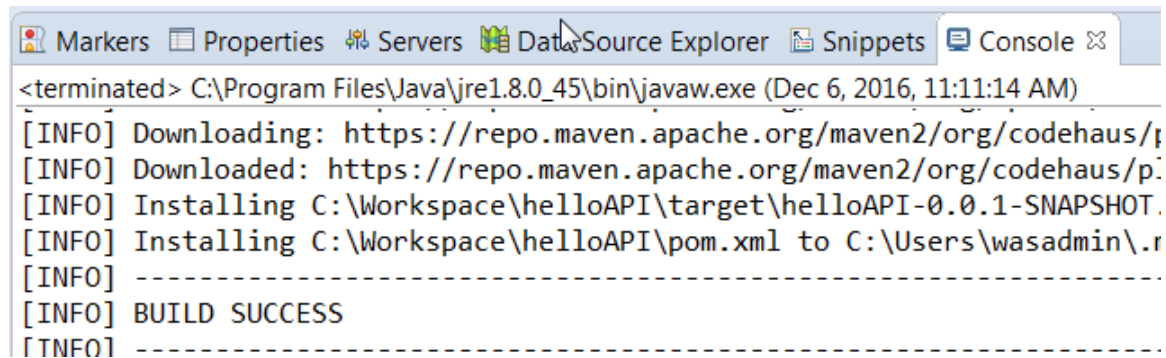
\_\_8. Save the file.

\_\_9. The **helloAPI** project node may show a small red 'x' to indicate an error. If so, right-click on the **helloAPI** project and then select **Maven** → **Update Project**, and then click **OK** in the resulting dialog.

\_\_10. In the **Project Explorer**, right-click on either the **helloAPI** project node or the 'pom.xml' file and then select **Run As** → **Maven Install**.

Note. If fails building try again and the second time should works.

The console should show a successful build. This ensures that we don't have any typos in the pom.xml entries we just did.



```
<terminated> C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Dec 6, 2016, 11:11:14 AM)
[INFO] Downloading: https://repo.maven.apache.org/maven2/org/codehaus/p
[INFO] Downloaded: https://repo.maven.apache.org/maven2/org/codehaus/p
[INFO] Installing C:\Workspace\helloAPI\target\helloAPI-0.0.1-SNAPSHOT
[INFO] Installing C:\Workspace\helloAPI\pom.xml to C:\Users\wasadmin\
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
```

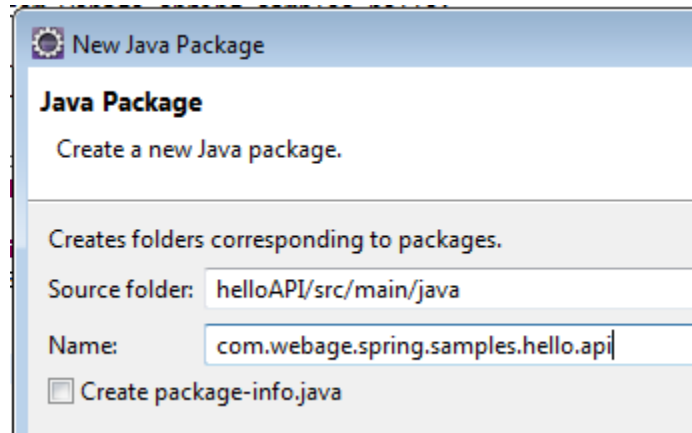
Now all we need to do is add a resource class and a response class.

## Part 4 - Implement the RESTful Service

In this part of the lab, we will create a response class and a RESTful resource class,

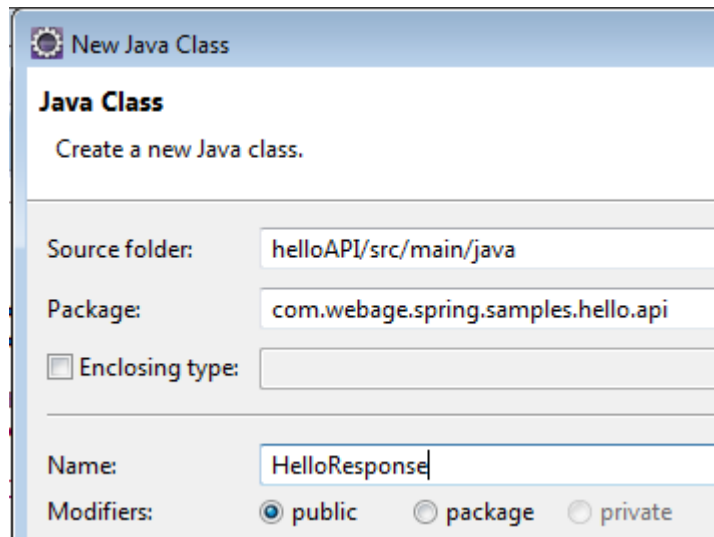
1. In the **Project Explorer**, right-click on **src/main/java** and then select **New** → **Package**.

2. Enter 'com.webage.spring.samples.hello.api' in the **Name** field, and then click **Finish**.



3. In the **Project Explorer**, right-click on the newly-created package and then select **New** → **Class**.

4. In the **New Java Class** dialog, enter 'HelloResponse' as the **Name**, and then click **Finish**.





\_\_5. Edit the body of the class so it reads as follows:

```
package com.webage.spring.samples.hello.api;

public class HelloResponse {
    String message;

    public HelloResponse(String message) {
        super();
        this.message = message;
    }

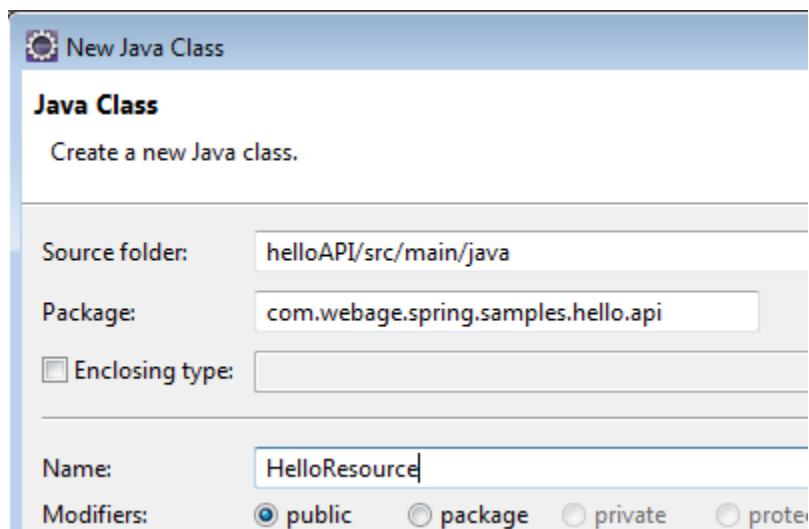
    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

\_\_6. Save the file.

\_\_7. In the **Project Explorer**, right-click on the 'com.webage.spring.samples.hello.api' package and then select **New** → **Class**.

\_\_8. In the **New Java Class** dialog, enter 'HelloResource' as the **Name**, and then click **Finish**.



\_\_9. Add the following 'getMessage' method inside the new class:

```
public HelloResponse getMessage() {  
    return new HelloResponse("Hello!");  
}
```

Spring Boot recognizes and configures the RESTful resource components by the annotations that we're about to place on the resource class that we just created.

\_\_10. Add the '@RestController' annotation to HelloResource, so it looks like:

```
@RestController  
public class HelloResource {
```

\_\_11. Add the '@GetMapping' annotation to the 'getMessage' method, so it looks like:

```
@GetMapping("/hello-message")  
public HelloResponse getMessage() {
```

\_\_12. Organize the imports by pressing **Ctrl-Shift-O**.

\_\_13. Save all files by pressing **Ctrl-Shift-S**.

\_\_14. In the **Project Explorer**, right-click on either the **helloAPI** project node or the 'pom.xml' file and then select **Run As** → **Maven Install**.

Note. If fails building try again and the second time should works.

The console should show a successful build.

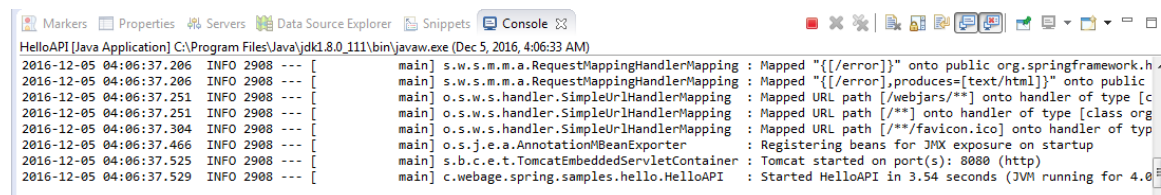
## Part 5 - Run and Test

That's all the components required to create a simple RESTful API with Spring Boot. Now let's fire it up and test it!

\_\_1. In the **Project Explorer**, right-click on the **HelloAPI** class and then select **Run as** → **Java Application**.

\_\_2. If the **Windows Security Alert** window pops up, click on **Allow Access**.

\_\_3. Watch the **Console** panel. At the bottom of it, you should see a message indicating that the 'HelloAPI' program has started successfully:

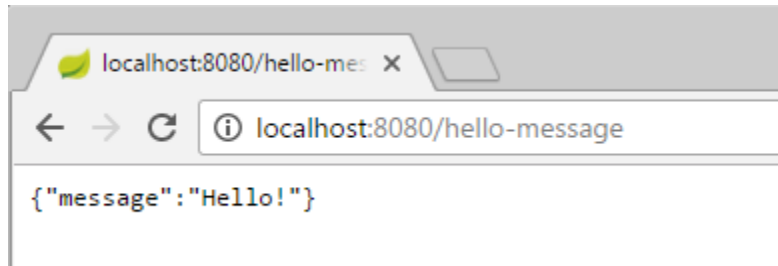


The screenshot shows the Eclipse IDE's Console panel. The title bar indicates the application is 'HelloAPI [Java Application]' running from 'C:\Program Files\Java\jdk1.8.0\_111\bin\javaw.exe' on 'Dec 5, 2016, 4:06:33 AM'. The console output shows a series of log messages from the Spring framework and Tomcat, including mapping of error handlers, URL paths for webjars and favicon, and the successful startup of the Tomcat server on port 8080. The final message states: 'Started HelloAPI in 3.54 seconds (JVM running for 4.0s)'.

\_\_4. Open the **Chrome** browser and enter the following URL in the location bar:

`http://localhost:8080/hello-message`

\_\_5. You should see the following response:



Notice that the response is in the form of a JSON object whose structure matches the 'HelloResponse' class contents.

\_\_6. Close the browser.

\_\_7. Click on the red 'Stop' button on the **Console** panel to stop the application.

\_\_8. Close all open files.

## Part 6 - Review

In this lab, we setup a rudimentary Spring Boot application. There are a few things you should notice:

- There was really very little code and configuration required to implement the very simple RESTful API.
- The resulting application runs in a standalone configuration without requiring a web or application server. It opens its own port on 8080 (we'll see later how to configure this port to any value you want).
- Although the Eclipse IDE is providing some nice features, like type-ahead support and automatic imports, the only tool we really need is a build tool that does dependency management (e.g. Apache Maven).

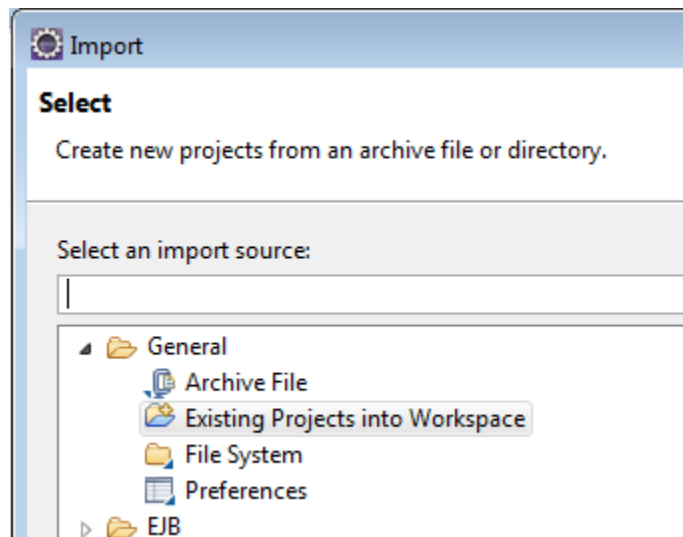
## Lab 2 - Use the Spring Web MVC Web Framework under Spring Boot

One of the many things Spring provides is a framework for web applications. This "Spring Web MVC" framework provides a lot of common features required in most web applications. This helps simplify the programming of web applications using Spring Web MVC so that the developers can focus on what the application is supposed to do instead of creating a framework to support web applications.

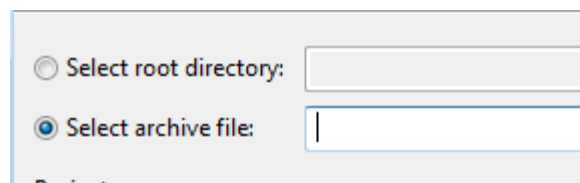
In this lab you will use some of these features of Spring Web MVC in the Spring Boot environment. This will be a simple application that manages "Purchase" data but will be enough to demonstrate the main features of the Spring Web MVC framework.

### Part 1 - Lab Setup

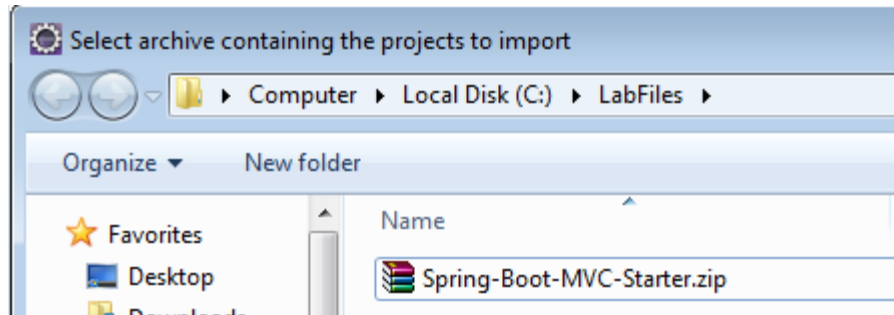
\_\_1. From Eclipse's main menu, click **File** → **Import** and then select **General** → **Existing Projects into Workspace**, and then click **Next**.



\_\_2. Click the radio button for **Select archive file**.



\_\_3. Click the **Browse** button and navigate to select **C:\LabFiles\Spring-Boot-MVC-Starter.zip** and then click **Open**.



\_\_4. On the **Import** dialog, leave the defaults as-is and click **Finish**.

## Part 2 - Examine the Spring Boot Configuration

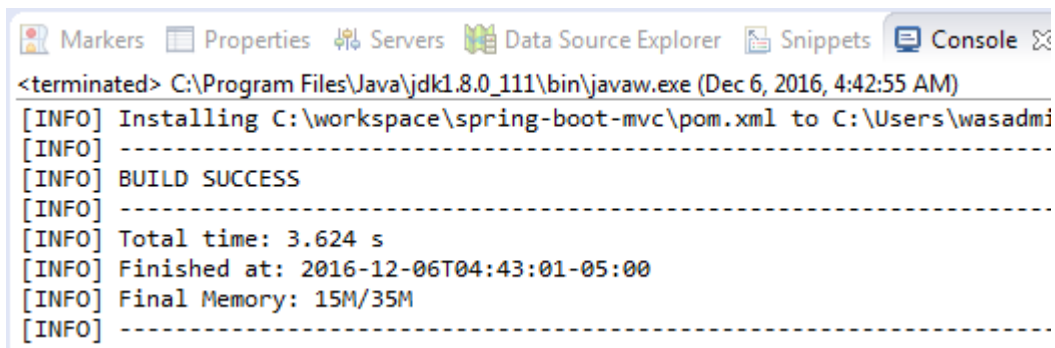
Have a look at the project that we've just imported. In particular, note the following:

- The 'pom.xml' file contains additional dependencies for the 'spring-boot-starter-web' artifact and the 'spring-boot-starter-thymeleaf' artifact in addition to the 'spring-boot-starter-parent' artifact.
- The 'src/main/resources' folder has a 'templates' folder that contains several html files. This folder is similar to the web root folder in a traditional JEE application.
- The 'src/main/java' folder contains a package, 'com.webage'. This package contains a class called 'App', that includes the Spring Boot startup code. All of the other components are in sub-packages of this package.
- The 'com.webage.dao' package contains an in-memory implementation of a storage repository for purchases.
- The 'com.webage.domain' package contains a domain object for purchases.
- The 'com.webage.service' package contains a service layer to look up purchases. This gives us a little bit of indirection where we can add business logic above the DAO classes.
- The 'com.webage.web' package has a class called 'PurchaseController' that acts as the target for web calls.

## Part 3 - Test Spring MVC Configuration

Before going too much further it will be good to test the current state of the project. Even though there is currently no other functionality you can test the request that should go to the 'index.jsp' file. This would test some of the configuration you just added to the 'spring-mvc.xml' file.

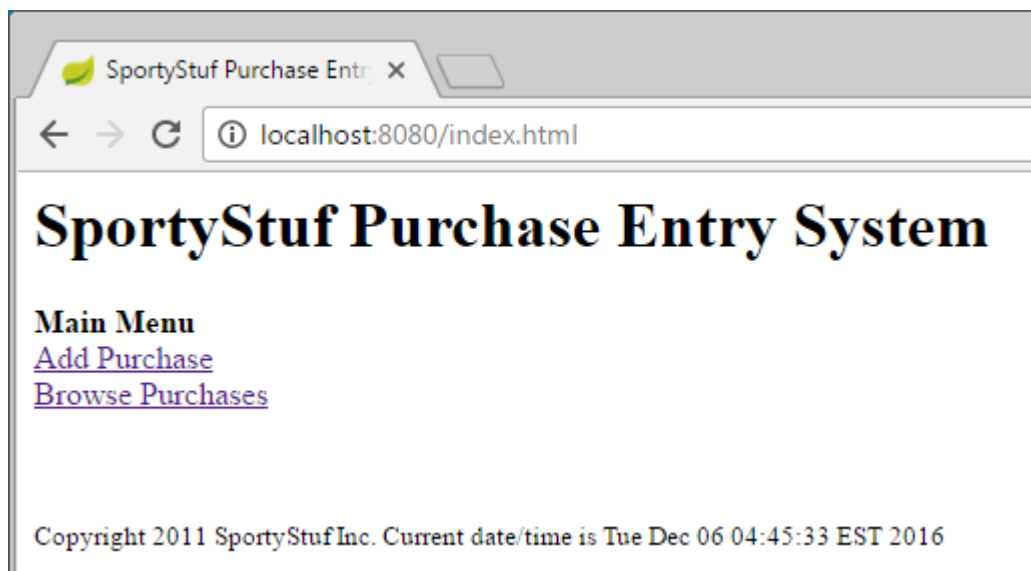
\_\_\_1. In the **Project Explorer** right-click on the **spring-boot-mvc** project and select **Run as** → **Maven install**. The build should run successfully. You may need to build twice since the first time sometimes doesn't build fine.



```
<terminated> C:\Program Files\Java\jdk1.8.0_111\bin\javaw.exe (Dec 6, 2016, 4:42:55 AM)
[INFO] Installing C:\workspace\spring-boot-mvc\pom.xml to C:\Users\wasadmi
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 3.624 s
[INFO] Finished at: 2016-12-06T04:43:01-05:00
[INFO] Final Memory: 15M/35M
[INFO] -----
```

\_\_\_2. Right-click on the class **App.java** in the **src/main/java/com.webage** package and then select **Run as** → **Java Application**. You should see a message in the console that the app has started.

\_\_\_3. Open a web browser and enter 'http://localhost:8080' into the location bar. You should see the browser redirect to '**localhost:8080/index.html**' and display the index page.



\_\_\_4. Click on the red stop button in the console panel to stop the Spring Boot application.



## Part 4 - Implement Browse Function

The first function that will be easiest to implement is the ability to browse the content of the purchase DAO. The underlying DAO function is already implemented and we just

need to add the Spring MVC web functionality in front of it.

\_\_ 1. Open the '**src/main/resources/templates/index.html**' file.

\_\_ 2. Find the first comment about a URL for the 'Add Purchase' link. Add the 'th:href' attribute of the link as shown in bold below. Be careful with the syntax as there are several double quotes and tag brackets.

```
<!-- Need URL for link -->
<a th:href="@{/addEditPurchase}">Add Purchase</a>
<br/>
```

**Note:** Sometimes in the html editor you will get mysterious red underlining pointing out an error that isn't there. If this happens, close all files, right click the file in question and select **Validate**, click **OK** on the box that comes up with the validation results, and then reopen the file. If the red underlining doesn't disappear when you do this, double check your syntax and then ask your instructor.

\_\_ 3. Find the second comment about needing a URL for the link for the 'Browse Purchases' link. Add the 'th:href' attribute of the link as shown in bold below. Be careful with the syntax as there are several double quotes and tag brackets.

```
<br/>
<!-- Need URL for link -->
<a th:href="@{/browse}">Browse Purchases</a>
```

\_\_ 4. Save the file and make sure there are no errors.

\_\_ 5. In the **Package Explorer** view, expand the following folders:

```
src/main/java -> com.webage.web
```

\_\_ 6. Open the **PurchaseController.java** file in the **com.webage.web** package by double clicking it. Notice that right now it has **@GetMapping** methods for '/index.html' and '/'. These mappings forward to the requisite templates. There is also an **@Controller** annotation on the class and an **@Autowired** injection of a 'PurchaseService' component. There's also a method annotated with '**@ModelAttribute**' that supplies the current date for use in the page footer.

\_\_7. Add the following new 'browsePurchases' method to the body of the class. The `@RequestMapping` annotation links this method to the '/browse' URL you used in the index.html page.

```
@RequestMapping("/browse")
public ModelAndView browsePurchases() {
    Collection<Purchase> list =
        purchaseService.findAllPurchases();
    return new ModelAndView("browsePurchases",
        "purchaseList", list);
}
```

**Note:** Also important is the ModelAndView object returned from the method. This will display the 'browsePurchases' view and make the list of purchases available as the 'purchaseList' variable in the view. You will see this used in the 'browsePurchases.jsp' file next.

\_\_8. Select **Source** → **Organize Imports**.

\_\_9. Save the file and make sure there are no errors.

\_\_10. Open the '**src/main/resources/templates/browsePurchases.html**' file.

\_\_11. Find the `<tr th:each="purchase : ${purchaseList}">` tag in the file about 2/3 of the way down. Notice that it will iterate over the 'purchaseList' that was made available by the controller. Each individual item will be available as the 'purchase' variable.

```
<tr th:each="purchase : ${purchaseList}">
    <!-- Add Edit link -->
    <td>Edit</td>
    <!-- Other purchase details -->
    <td></td>
    <td></td>
    <td></td>
    <td></td>
</tr>
```

\_\_12. Within the four columns that are currently empty **add** the following syntax for various expressions and a date format, all in Thymeleaf format.

```
<!-- Other purchase details -->
<td th:text="${purchase.id}"></td>
<td th:text="${purchase.customerName}"></td>
<td th:text='${#{#dates.format(purchase.purchaseDate, "MMM d,
yyyy")}'></td>
<td th:text="${purchase.product}"></td>
```

\_\_13. Save the file and make sure there are no errors.



## Part 5 - Test Browse Function

\_\_1. Run 'Maven install' and then execute the 'App.java' class using the same technique as in the previous lab part.

\_\_2. Open a web browser to:

**http://localhost:8080/**

\_\_3. Click on the '**Browse Purchases**' link and be sure you get the list of the three purchases currently in the database.

### Browse Purchases

[Add Purchase](#)

	Purchase Id	Customer Name	Purchase Date	Product
Edit	1	Susan	May 12, 2010	Mountain Bike
Edit	2	Bob	Apr 30, 2010	Football
Edit	3	Jill	Jun 5, 2010	Kayak

[Back to Main Menu](#)

\_\_4. Click on the '**Back to Main Menu**' link and make sure the home page is displayed.

\_\_5. Close the browser.

\_\_6. Click on the red stop button in the console panel to stop the Spring Boot application.



\_\_7. Close all open files.

## Part 6 - Review

Spring MVC has many useful features that simplify web application programming. Spring MVC can do things like register URLs that are requested with Controller methods or view pages and bind the properties of a Java object to the fields on a form. Spring MVC supports several view templating technologies; we had a quick look at the 'Thymeleaf' templates.

Although this lab only showed a brief part of what is possible with Spring MVC you got a sense for the "**Model, View, Controller**" framework that is what makes up Spring MVC.

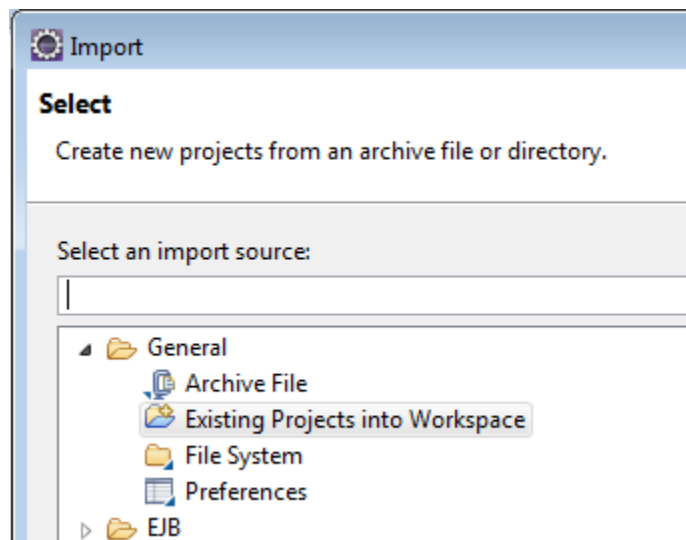
## Lab 3 - Use the Spring JDBCTemplate under Spring Boot

Any of Spring's data access techniques can be used with Spring Boot. For convenience, Spring Boot sets up an embedded database by default, which you can override with an external database later on. This is useful for testing and early development on an application.

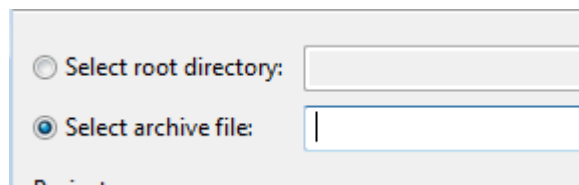
In this lab you will implement a Data Access Object by creating SQL queries and issuing them with a JDBCTemplate that is autowired by Spring.

### Part 1 - Import the Starter Project

\_\_1. From Eclipse's main menu, click **File** → **Import** and then select **General** → **Existing Projects into Workspace**, and then click **Next**.

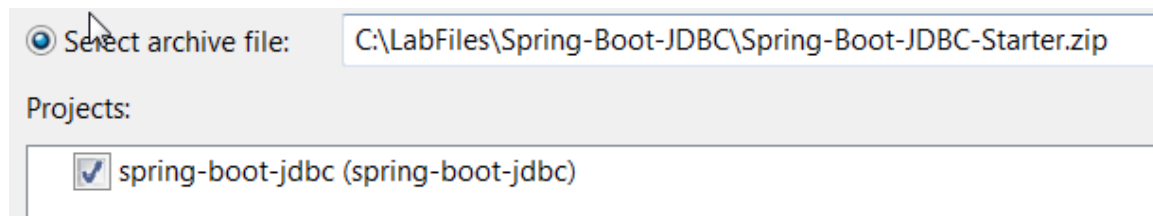


\_\_2. Click the radio button for **Select archive file**:



\_\_3. Click the **Browse** button and navigate to select **C:\LabFiles\Spring-Boot-JDBC\Spring-Boot-JDBC-Starter.zip** and then click **Open**.

\_\_4. On the **Import** dialog, leave the defaults as-is and click **Finish**.



## Part 2 - Enable the Default Embedded Database

Spring will auto-configure a database for us, and automatically load a data set if necessary. All we need to do is give it the correct setup information.

\_\_1. In the **Project Explorer**, locate the 'pom.xml' file for **spring-boot-jdbc** project and double-click it to open it.

\_\_2. Select the pom.xml tab.

\_\_3. Add the following dependency into the '<dependencies>' element:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
  <groupId>org.hsqldb</groupId>
  <artifactId>hsqldb</artifactId>
</dependency>
```

\_\_4. Save the file. You will notice that the errors in the project have gone.

\_\_5. Locate the files 'C:\LabFiles\Spring-Boot-JDBC\schema.sql' and 'C:\LabFiles\Spring-Boot-JDBC\data.sql'.

\_\_6. Copy both these files into the '**src/main/resources**' folder in Eclipse. These files are used at startup time to initialize the in-memory HSQLDB database. Overwrite the existing files.

\_\_7. Right click on the **spring-boot-jdbc** project and select **Run as** → **Maven install** to Run a Maven install and ensure that there are no errors. This will confirm that you have no typos in the dependencies. You may need to run the Maven Install two times to get the Build Success message.

### Part 3 - Configure the JDBCTemplate

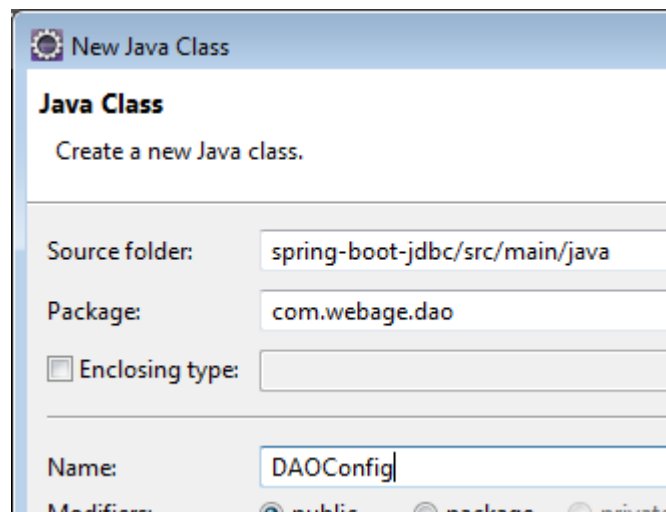
Because we added the dependency for HSQLDB to 'pom.xml', Spring Boot will automatically create a DataSource object for an embedded, memory-based instance of HSQLDB. Also, when the application starts up, Spring Boot will initialize the schema of the embedded database using the 'schema.sql' file that is in the classpath, and then run the 'data.sql' file that's also in the classpath.

To use the database from our Data Access Object, we'll need to configure a JDBCTemplate object that can be injected into the DAO. We'll do that by creating a class and annotating it with '@Configuration'. This is an example of Spring's 'Java-based configuration' mechanism. Spring will find the annotated class in the classpath and use it to instantiate the associated beans.

\_\_\_ 1. Locate the package 'com.webage.dao' in the **Project Explorer** under '**spring-boot-jdbc/src/main/java**'.

\_\_\_ 2. Right-click on the 'com.webage.dao' package and select **New** → **Class**.

\_\_\_ 3. Enter 'DAOConfig' as the new class name and then click **Finish**.



\_\_\_ 4. Add the annotation '@Configuration' to the class, as shown below:

```
@Configuration
public class DAOConfig {
```

\_\_\_ 5. Organize the imports by pressing **Ctrl-Shift-O**.

\_\_6. Inside the class, create a method 'jdbcTemplate()' as shown below:

```
@Bean JdbcTemplate jdbcTemplate(dataSource ds) {  
    return new JdbcTemplate(ds);  
}
```

\_\_7. Organize the imports by pressing **Ctrl-Shift-O**. Select **javax.sql.DataSource**.

\_\_8. Save the file.

The method we added defines a bean called 'jdbcTemplate' that can be injected into our DAO class.

## Part 4 - Complete the DAO Class

Now that we have the configuration complete, all we need to do is flesh out the implementation of the DAO class to actually perform the query. We'll use some convenience classes provided by the Spring Framework to make this relatively painless.

\_\_1. Locate the class called 'JDBCPurchaseDAO' in the 'com.webage.dao' package. Double-click on the class to open it.

\_\_2. Look for the method called that currently looks like below:

```
@Override  
public Collection<Purchase> getAllPurchases() {  
    // Replace this statement with the call to jdbcTemplate.  
    return null  
}
```

\_\_3. Replace the line 'return null;' with a call to jdbcTemplate, so that the method appears as :

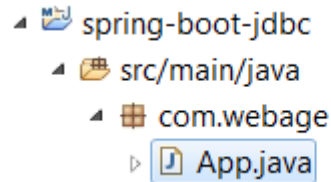
```
@Override  
public Collection<Purchase> getAllPurchases() {  
    // Replace this statement with the call to jdbcTemplate.  
    return jdbcTemplate.query("Select * from PURCHASE", new  
        BeanPropertyRowMapper<Purchase>(Purchase.class));  
}
```

Here we're using Spring's 'BeanPropertyRowMapper' class to automatically map the database rows to instances of the 'Purchase' class based on the column names. If you look in the 'schema.sql' file, you'll notice that the column names match the property names that are used in the 'Purchase' class. That correspondence allows us to use this convenience class.

\_\_4. Save all files by pressing **Ctrl-Shift-S**.

## Part 5 - Compile and Test

- \_\_\_ 1. Using the same technique as in previous labs, run a 'Maven install' operation.
- \_\_\_ 2. Run the 'App' class as a Java application.



- \_\_\_ 3. Open a browser and enter the following url:

`http://localhost:8080/browse`

- \_\_\_ 4. You should see the results of our database query.

## Browse Purchases

[Add Purchase](#)

	Purchase Id	Customer Name	Purchase Date	Product
<a href="#">Edit</a>	1	Bruce	May 12, 2010	Mountain Bike
<a href="#">Edit</a>	2	Paul	Apr 30, 2010	Football
<a href="#">Edit</a>	3	Rick	Jun 5, 2010	Kayak

[Back to Main Menu](#)

Copyright 2011 SportyStuf Inc. Current date/time is Tue Dec 06 13:55:53 EST 2016

- \_\_\_ 5. Close the browser.
- \_\_\_ 6. Close all open files.
- \_\_\_ 7. Click on the red stop button in the console panel to stop the Spring Boot application:



## **Part 6 - Review**

We used the convenient embedded database setup in this lab to demonstrate the use of the Spring JdbcTemplate to carry out a query against a SQL DataSource.

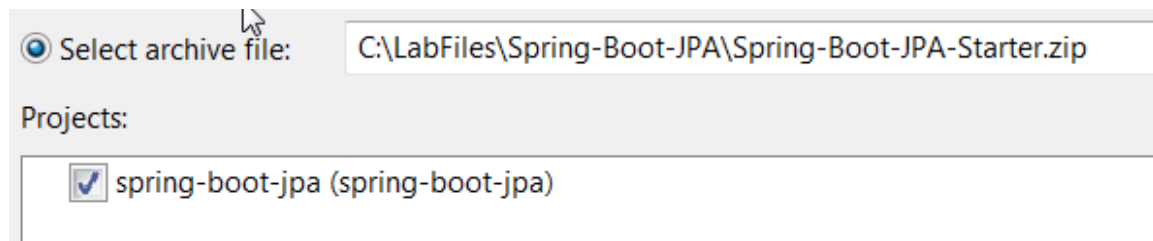
## Lab 4 - Use the Spring Data JPA under Spring Boot

Java Persistence Architecture, or JPA simplifies data access by automatically generating SQL queries to manage the storage and retrieval of Java objects. Spring Data takes that idea one step farther to automatically generate data access or "Repository" classes for Java objects. All we need to do is make sure our Java objects are annotated correctly to contain the additional metadata required for database storage. Also, of course, we need to setup the software infrastructure.

In this lab you will annotate a set of domain objects and then use Spring Data to implement a storage repository for those objects.

### Part 1 - Import the Starter Project

- \_\_1. From Eclipse's main menu, click **File** → **Import** and then select **General** → **Existing Projects into Workspace**, and then click **Next**.
- \_\_2. Click the radio button for **Select archive file**.
- \_\_3. Click the **Browse** button and navigate to select **C:\LabFiles\Spring-Boot-JPA\Spring-Boot-JPA-Starter.zip** and then click **Open**.
- \_\_4. On the **Import** dialog, leave the defaults as-is and click **Finish**.



### Part 2 - Examine the Starter Project

- \_\_1. Expand the **spring-boot-jpa** project.
- \_\_2. Open pom.xml and select the pom.xml tab.

There are a few items already setup in the starter project that you should take note of for your own projects:



- 'pom.xml' contains two dependencies that are particularly important to this project:

```
<dependency>
  <groupId>org.hsqldb</groupId>
  <artifactId>hsqldb</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

- These dependencies pull in the components required for Spring Boot to implement the Spring Data framework, and also to use the embedded HSQLDB instance for testing.
- In 'src/main/resources', there are two files, 'data.sql' and 'schema.sql', which establish the database schema and initial data load for the embedded database that we'll use for testing. These files end up being assembled to the root of the classpath, where Spring Boot can load them at startup.
- Also in 'src/main/resources', there is a file called 'application.properties'. This file contains one line:

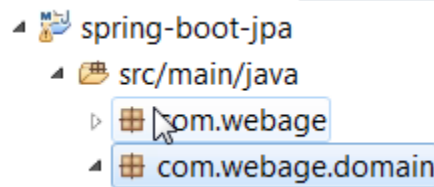
```
spring.jpa.hibernate.ddl-auto=false
```

- The line disables Hibernate's automatic schema generation, which would overwrite our test data if we left it enabled.
- The starter project contains a pair of domain classes that we will use as a starting point for our repository.
- The starter project also contains a rudimentary user interface for our demonstration program.

### Part 3 - Annotate the Domain Classes

We have a pair of domain classes, 'Customer' and 'Purchase' that are meant to model a set of purchases that might be made through an online store of some kind. There is a "Many-to-One" relationship between these classes; many purchases might be made by one customer. We haven't modeled the reverse relationship, although that might also be useful.

\_\_1. Locate the package 'com.webage.domain' in the **Project Explorer**.



\_\_2. Locate the **Customer** class inside 'com.webage.domain' and double-click on the class to open it.

\_\_3. Add the '@Entity' and '@Table' annotations to the class as shown below. These annotations designate that the class should be managed by JPA and tell JPA what database table name to use for the class:

```
@Entity
@Table(name="CUSTOMERS")
public class Customer {
```

\_\_4. Add annotations to the 'id' field so it appears as below. These annotations mark the 'id' field as the primary key of 'Customer' and tell JPA how it's going to be generated.

```
@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
long id;
```

\_\_5. The 'name' field in 'Customer' is modeled by a column called 'CUSTOMER\_NAME' in the database table that we are using. So, the default mapping of field name to column name will not work in this case. Annotate the field as shown below to override the default name mapping:

```
@Column(name="CUSTOMER_NAME")
String name;
```

\_\_6. Organize the imports by pressing **Ctrl-Shift-O**. Select **javax.persistence.Entity** and **javax.persistence.Id**.

\_\_7. Save the file.

\_\_8. Locate the **Purchase** class inside 'com.webage.domain' and double-click on the class to open it.

\_\_9. Add the '@Entity' and '@Table' annotations to the class as shown below. These annotations designate that the class should be managed by JPA and tell JPA what database table name to use for the class:

```
@Entity
@Table(name="PURCHASES")
public class Purchase {
```

\_\_10. Add annotations to the 'id' field so it appears as below. These annotations mark the 'id' field as the primary key of 'Customer' and tell JPA how it's going to be generated.

```
@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
private int id;
```

\_\_11. Annotate the 'customer' field as shown below to tell JPA that it should load the value from an associated table.

```
@ManyToOne
private Customer customer;
```

\_\_12. Organize the imports by pressing **Ctrl-Shift-O**. Select **javax.persistence.Entity**, **javax.persistence.Table** and **javax.persistence.Id**.

\_\_13. Save the file.

We now have a set of domain objects that should be recognized as JPA Entities that can be managed by JPA.

## Part 4 - Create the Repository Interfaces

This is where we see the magic of the Spring Data JPA framework!

The starter project has two interfaces declared in the 'com.webage.repository' package. Spring Data uses these interfaces to automatically generate classes that implement the repository functionality. Let's have a look at one of them

\_\_1. Locate the interface called 'PurchaseRepository' in the 'com.webage.repository' package. Double-click on the class to open it.

For convenience, the contents are reproduced below:

```
package com.webage.repository;

import org.springframework.data.repository.CrudRepository;

import com.webage.domain.Purchase;

public interface PurchaseRepository extends CrudRepository<Purchase,
Long> {

}
```

As you can see, there's really not much to this repository interface. It extends a standard interface called 'CrudRepository' that includes standard Create, Retrieve, Update and Delete methods. It uses Java generic type definitions to indicate what data type the repository holds, and the data type of the primary identifier field.

That information is enough for Spring Data to fully implement this interface. We don't need to write any access code at all!

If you take a look at the 'CustomerRepository' interface you'll find a similar declaration based on the Customer domain class.

## Part 5 - Observe the Usage

1. Locate the 'PurchaseServiceImpl' class in the 'com.webage.services' package. Double-click on the class to open it. For convenience, the contents are shown below:

```
package com.webage.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.webage.domain.Purchase;
import com.webage.repository.PurchaseRepository;

@Service
public class PurchaseServiceImpl implements PurchaseService {
    @Autowired
    private PurchaseRepository repo;

    public void savePurchase(Purchase purchase) {
        repo.save(purchase);
    }
}
```

```

        public Iterable<Purchase> findAllPurchases() {
            return repo.findAll();
        }

        public Purchase findPurchaseById(long id) {
            return repo.findOne(id);
        }
    }
}

```

\_\_2. Notice the '@Autowired' field for the PurchaseRepository. Spring Data will automatically create a class that implements the PurchaseRepository interface and plug it in to the instance of 'PurchaseServiceImpl'.

\_\_3. Notice that the service calls the methods 'save(...)', 'findAll()' and 'findOne(...)' on the repository interface. These methods will be implemented by Spring Data's automatic proxy, so as to provide the expected functionality.

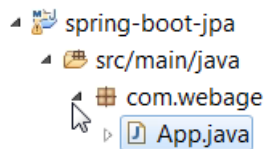
## Part 6 - Compile and Test

\_\_1. Right click **spring-boot-jpa** and select **Maven → Update Project**.

\_\_2. Make sure **spring-boot-jpa** is selected and click OK.

\_\_3. Right click on the **spring-boot-jpa** project and select **Run as → Maven install** to Run a Maven install and ensure that there are no errors. You may need to run the Maven Install two times to get the Build Success message.

\_\_4. Run the 'App' class as a Java application.



\_\_5. Open a browser and enter the following url:

```
http://localhost:8080/browse
```

\_\_6. You should see the results of our database query.

## Browse Purchases

[Add Purchase](#)

	Purchase Id	Customer Name	Purchase Date	Product
Edit	1	Bruce	May 12, 2010	Mountain Bike
Edit	2	Paul	Apr 30, 2010	Football
Edit	3	Rick	Jun 5, 2010	Kayak

[Back to Main Menu](#)

Copyright 2011 SportyStuf Inc. Current date/time is Tue Dec 06 13:55:53 EST 2016

\_\_7. Close the browser.

\_\_8. Close all open files.

\_\_9. Click on the red stop button in the console panel to stop the Spring Boot application:



## Part 7 - Review

In this lab, we used Spring Data's functionality combined with Spring JPA to implement a data repository very rapidly, with a minimum of effort.

## Lab 5 - Learning the MongoDB Lab Environment

MongoDB is an open source document-oriented NoSQL database system written in C++. MongoDB stores documents in a JSON-like binary format called BSON that MongoDB uses for both data storage and network transfer. BSON is not a BLOB-like binary as it is traversable and, like the regular JSON, supports object and array embedding.

In this lab, we will show how to set up, run, and access an instance of MongoDB on your computer.

### Part 1 - Understanding the Main Files

The MongoDB system has already been setup for you in the **C:\Software\mongodb** folder. We will refer to this location as **MONGO\_HOME**. The system binaries are located in the **MONGO\_HOME\bin** folder.

**Note:** MongoDB is a self-contained system with no system dependencies. It is installed by unzipping the distribution archive in the directory of your choice.

The MongoDB database process is launched by a call to the **MONGO\_HOME\bin\mongod.exe** file.

When the process has been properly initialized, you can launch the MongoDB shell **MONGO\_HOME\bin\mongo.exe** which, by default, will try to connect to mongod.exe on the localhost interface and port 27017.

### Part 2 - Creating the Data Folder

MongoDB requires a data folder to store its database files. By default, on Windows platform, the location for the MongoDB data directory is **C:\data\db**. You can use another location, but in this case, you need to let MongoDB know where this directory is (which is done by passing the location as a *--dbpath* configuration option, e.g.

`mongod.exe --dbpath c:\projectX\data`).

\_\_1. Open a command prompt window and type in the following command at the prompt and press **ENTER** (execute the command):

```
dir c:\data\db
```

## Part 3 - Launching MongoDB

\_\_1. In the command prompt window, execute the following command:

```
cd C:\Software\mongodb\bin
```

This command will change directory to MONGO\_HOME\bin where all MongoDB binaries are located.

\_\_2. Execute the following command:

```
title MONGOD
```

This command will set up the title of the window for ease of reference.

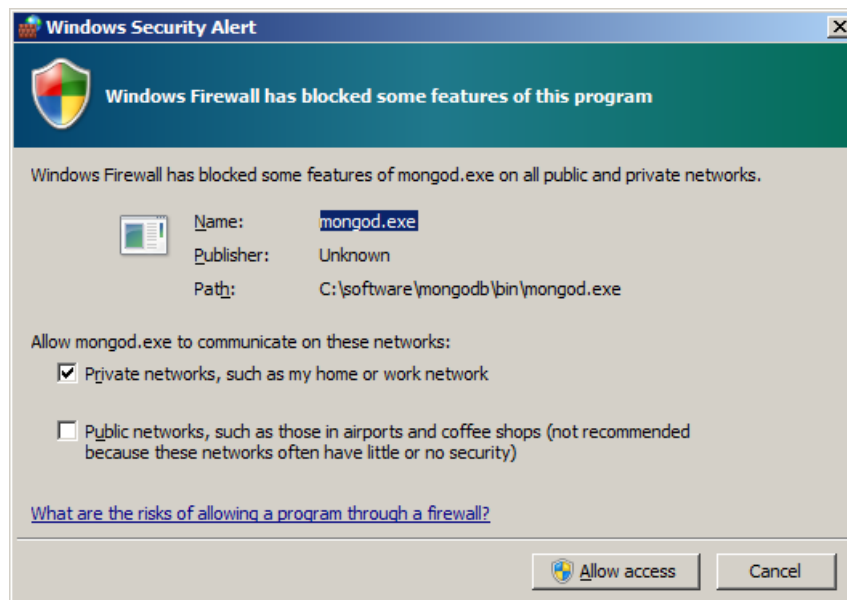


\_\_3. Execute the following command:

```
mongod --rest
```

This command will start the MongoDB process and will also activate the RESTful endpoint on the admin web console.

If you see the Windows Security Alert below then click the **Allow access** button:





On successful start up, you should see the following diagnostic messages (last in the console output).

```
2018-02-27T11:01:08.096-0500 I CONTROL [initandlisten]
2018-02-27T11:01:08.097-0500 I CONTROL [initandlisten] Hotfix KB2731284 or later update is not installed, will zero-out data files.
2018-02-27T11:01:08.097-0500 I CONTROL [initandlisten]
2018-02-27T11:01:08.099-0500 I NETWORK [websvr] admin web console waiting for connections on port 28017
2018-02-27T11:01:08.940-0500 I FTDC [initandlisten] Initializing full-time diagnostic data capture with directory 'C:/data/db/diagnostic.data'
2018-02-27T11:01:08.940-0500 I NETWORK [thread1] waiting for connections on port 27017
```

**Note:** The sequence of the messages may be reverse, depending on which component has completed its initialization first.

Port 27017 is used for local process connections (our shell will connect to MongoDB via this port), and port 28017 is used to access admin web console using browser.

In subsequent labs, we will be referring to the above steps as ***Start the MongoDB process.***

## Part 4 - Verifying Access to the Admin Web Console

\_\_1. Open Google Chrome browser and navigate to **http://localhost:28017**

The Admin Web Console should open. The version of mongo may be different.

### **mongod WA2706**

[List all commands](#) | [Replica set status](#)

Commands: [isMaster](#) [listDatabases](#) [serverStatus](#) [top](#) [buildInfo](#) |

```
db version v3.4.13-20-g1edf4ba
git hash: 1edf4baa102a0a26dad5730ebafb5dfcb1714bed
uptime: 24 seconds
```

The admin console is rather minimalistic in its UI design and provides a view of a variety of system parameters.

## Part 5 - Using the MongoDB Shell

\_\_1. Open another command prompt window and execute the following command at the prompt:

```
cd C:\Software\mongodb\bin
```

We will use this command window to launch the MongoDB shell to execute commands against MongoDB.

\_\_2. Execute the following command:

```
title MONGO SHELL
```

This command will set up the title of the window as MONGO SHELL.

\_\_3. Execute this command:

```
mongo
```

This command will start the MongoDB shell, connect to the MongoDB process on port 27017 and return the command prompt '>'.

You may see some warnings, for now you can safely ignore them.

**Note:** You quit the shell by typing the *exit* command.

```
C:\Software\mongodb\bin>mongo
MongoDB shell version v3.4.13-20-g1edf4ba
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 3.4.13-20-g1edf4ba
Server has startup warnings:
2018-02-27T11:01:07.611-0500 I CONTROL [main] ** WARNING: --rest is specified
ce.
2018-02-27T11:01:07.611-0500 I CONTROL [main] ** enabling http interface
2018-02-27T11:01:08.094-0500 I CONTROL [initandlisten]
2018-02-27T11:01:08.096-0500 I CONTROL [initandlisten] ** WARNING: Access control
r the database.
2018-02-27T11:01:08.096-0500 I CONTROL [initandlisten] ** Read and write
d configuration is unrestricted.
2018-02-27T11:01:08.096-0500 I CONTROL [initandlisten]
2018-02-27T11:01:08.097-0500 I CONTROL [initandlisten] Hotfix KB2731284 or later
alled, will zero-out data files.
2018-02-27T11:01:08.097-0500 I CONTROL [initandlisten]
> _
```

The version of mongo may be different.

By default, you will be connected to the *test* db. You can easily create your own databases, if needed.

In subsequent labs, we will be referring to the above steps as ***Start the MongoDB Admin shell***.

We will explore the MongoDB shell in more detail in one of the subsequent labs.

We are almost done in this lab.

## Part 6 - Clean-Up

\_\_1. In the MONGO SHELL window, execute this command:

**exit**

This command will stop the current shell session.

\_\_2. Close the MongoDB Shell terminal.

We will refer to the above steps related to shutting down the MongoDB shell as ***Shut down the MongoDB Admin shell***.

\_\_3. In the MONGODB command window, press **Ctrl-C** (you may need to repeat this command again to terminate the process for sure).

This command will flush the database buffers, remove database locks and shutdown the MongoDB process.

\_\_4. Close the MONGODB terminal.

\_\_5. Close the browser.

We will refer to the above steps related to shutting down the MongoDB process as ***Shut down MongoDB***.

This is the last step in this lab.

## Part 7 - Review

In this lab, we learned about the MongoDB lab environment.

## Lab 6 - Spring Data with MongoDB

In this lab we will use MongoDB with a SpringBoot microservice. We will use the **MongoRepository** interface and letting Spring create the query methods we need. We will map the **MongoRepository** to a MongoDB database called **inventory**. We will also create a car class and map it to a MongoDB collection called **vehicles**.

Our microservice class will expose RESTful methods to add a car, delete all cars, get a specific car, get all cars and search for cars by specific criteria.

### Part 1 - Setup

In this part we will perform the basic setup for the lab.

- \_\_ 1. Make sure that no Spring Boot applications are running.
- \_\_ 2. Open a DOS shell and start the **MongoDB server** as you learned in a previous lab.
- \_\_ 3. Open another DOS shell and start the **Mongo shell** as you learned in a previous lab.
- \_\_ 4. Examine the **C:\Labfiles\Spring-Boot-JPA-Mongo** directory, you should the **pom.xml** file and the src directory.

NOTE: for simplicity in this lab we will use the following convention

Shortcut	Path
<mongo>	C:\Labfiles\Spring-Boot-JPA-Mongo

### Part 2 - The Car Class

In this part we will complete the **Car** class. The Car class is a JavaBean that represents cars in the system. We will use an annotation to map the Car class to the **vehicles** collection in our MongoDB database.

- \_\_ 1. Open <mongo>\src\main\java\webage\data\Car.java in a text editor or IDE.  
The Car class has four properties: **id**, **make**, **model**, **year**  
Notice that the **id** property is annotated with **@Id** indicating that this is the id property for MongoDB
- \_\_ 2. Add standard Getters and Setters to the **Car** class.
- \_\_ 3. Add a no-argument constructor.
- \_\_ 4. Add the **@Document** annotation to the **Car** class.
- \_\_ 5. Map the **Car** class by setting the **collection** property of the annotation to **"vehicles"**.

\_\_6. Your code should look like this:

```
package webage.data;

import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

@Document(collection="vehicles")
public class Car {

    @Id
    private String id;

    private String make;
    private String model;
    private int year;

    public Car() {}

    public String getId() {
        return id;
    }

    public String getMake() {
        return make;
    }

    public void setMake(String make) {
        this.make = make;
    }

    public String getModel() {
        return model;
    }

    public void setModel(String model) {
        this.model = model;
    }

    public int getYear() {
        return year;
    }

    public void setYear(int year) {
        this.year = year;
    }

}
```

\_\_7. Save the file.

## Part 3 - The MongoRepository Interface

In this part you will create an interface that extends the **MongoRepository**. This interface is used by Spring to generate code to access the MongoDB database. We've already mapped the **Car** bean to a collection, we will also map the application to a database.

\_\_\_ 1. Examine `<mongo>\src\main\java\webage\data\CarRepository.java` in text editor.

\_\_\_ 2. Add the interface-level annotation **@RepositoryRestResource**

\_\_\_ 3. Modify the **CarRepository** interface so that it extends **MongoRepository<T, ID>**

The type parameters **T** and **ID** represent the data type stored in the collection and the type of the **id** field. Set these to **Car** and **String** respectively.

\_\_\_ 4. Examine the **findByYearAndMakeAndModel** method.

The naming convention includes the fields used in the search.

\_\_\_ 5. Complete the interface by following the naming convention and adding declarations for the following methods:

```
findByYearAndMake  
findByYearAndModel  
findByMakeAndModel  
findByModel  
findByMake  
findByYear
```

\_\_\_ 6. Your code should look like this:

```
package webage.data;  
  
import java.util.List;  
import java.util.Optional;  
  
import org.springframework.data.mongodb.repository.MongoRepository;  
import org.springframework.data.repository.query.Param;  
import  
org.springframework.data.rest.core.annotation.RepositoryRestResource;  
  
@RepositoryRestResource  
public interface CarRepository extends MongoRepository<Car, String> {  
  
    List<Car> findByYearAndMakeAndModel(  
        @Param("year") int year,  
        @Param("make") String make,  
        @Param("model") String model);  
  
    List<Car> findByYearAndMake(  
        @Param("year") int year,
```

```

        @Param("make") String make);

List<Car> findByYearAndModel(
    @Param("year") int year,
    @Param("model") String model);

List<Car> findByMakeAndModel(
    @Param("make") String make,
    @Param("model") String model);

List<Car> findByModel(@Param("model") String model);
List<Car> findByMake(@Param("make") String make);
List<Car> findByYear(@Param("year") int year);
}

```

\_\_7. Save the file.

\_\_8. Open <mongo>\src\main\resources\application.properties in a text editor

\_\_9. Add the following property to map the application to the **inventory** database

```
spring.data.mongodb.database=inventory
```

\_\_10. Save the file.

## Part 4 - Complete the Microservice

In this part we will complete a microservice that adds cars to the database, retrieves cars from the database and clears all cars from the database.

We will add code to the **deleteAll()** method that deletes all cars in the database.

The **findWithParameters()** method takes three optional parameters: **year**, **model** and **make**. The method should call the various **findBy...** methods of the **CarRepository**.

\_\_1. Open <mongo>\src\main\java\webage\MongoServiceApplication.java in a text editor.

\_\_2. Add the following code to the **deleteAll()** method as shown in bold.

```

@RequestMapping(method=RequestMethod.DELETE)
public ResponseEntity deleteAll() {

    // ADD CODE TO DELETE ALL CARS
    carRepository.deleteAll();
}

```

```

    return new ResponseEntity(HttpStatus.OK);
}

```

Notice that we never explicitly declared a **deleteAll()** method in the **CarRepository**. This method is inherited from **MongoRepository** and **CrudRepository**.

Examine the **findWithParameters()** method. If the **year**, **make** and **model** parameters are passed in the query string. The method calls the **carRepository.findByYearMakeAndModelMethod()**.

\_\_3. Add **else if** clauses to handle all the combinations of parameters and call the appropriate **findBy...** methods below the *//INSERT 'else if's HERE* line.

```

// INSERT 'else if's HERE
else if ( year.isPresent() && make.isPresent() ) {

    cars = carRepository.findByYearAndMake(year.get(), make.get());

} else if ( year.isPresent() && model.isPresent() ) {

    cars = carRepository.findByYearAndModel(year.get(), model.get());

} else if ( make.isPresent() && model.isPresent() ) {

    cars = carRepository.findByMakeAndModel( make.get(), model.get());

} else if( year.isPresent() ) {

    cars = carRepository.findByYear(year.get());

} else if ( make.isPresent() ) {

    cars = carRepository.findByMake(make.get());

} else if ( model.isPresent() ) {

    cars = carRepository.findByModel(model.get());

} else {

    cars = carRepository.findAll();

}

```

\_\_4. Save the file.



## Part 5 - Test

In this part we will build and run the application. Next we will use Postman to add some Cars to the database. Then we will query the database to retrieve the cars using different criteria.

- \_\_ 1. Open a DOS shell to **<mongo>**
- \_\_ 2. Run the application by entering the following command

```
mvn spring-boot:run
```

- \_\_ 3. The application should start successfully.

```
sourceInformation>
2018-02-27 11:22:37.206 INFO 5548 --- [main] o.s.d.r.w.BasePathAwareHandlerMapping :
Mapped "[/profile].methods=[OPTIONS]" onto public org.springframework.http.ResponseEntity<?> org.sprin
gframework.data.rest.webmvc.ProfileController.profileOptions()
2018-02-27 11:22:37.207 INFO 5548 --- [main] o.s.d.r.w.BasePathAwareHandlerMapping :
Mapped "[/profile].methods=[GET]" onto org.springframework.http.ResponseEntity<org.springframework.hat
eoas.ResourceSupport> org.springframework.data.rest.webmvc.ProfileController.listAllFormsOfMetadata()
2018-02-27 11:22:37.370 INFO 5548 --- [main] o.s.j.e.a.AnnotationMBeanExporter :
Registering beans for JMX exposure on startup
2018-02-27 11:22:37.433 INFO 5548 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer :
Tomcat started on port(s): 8080 (http)
2018-02-27 11:22:37.438 INFO 5548 --- [main] webage.MongoServiceApplication :
Started MongoServiceApplication in 3.74 seconds (JVM running for 15.426)
```

- \_\_ 4. Open Postman.
- \_\_ 5. Select **POST**.
- \_\_ 6. Enter the **POST** URL as **http://localhost:8080/cars/add**
- \_\_ 7. Click **Headers**.
- \_\_ 8. Enter as key **Content-Type**.
- \_\_ 9. Enter as value **application/json**.

Authorization	Headers (1)	Body ●	Pre-request Script	Tests				
	<table><tr><th>Key</th><th>Value</th></tr><tr><td><input checked="" type="checkbox"/> Content-Type</td><td>application/json</td></tr></table>	Key	Value	<input checked="" type="checkbox"/> Content-Type	application/json			
Key	Value							
<input checked="" type="checkbox"/> Content-Type	application/json							

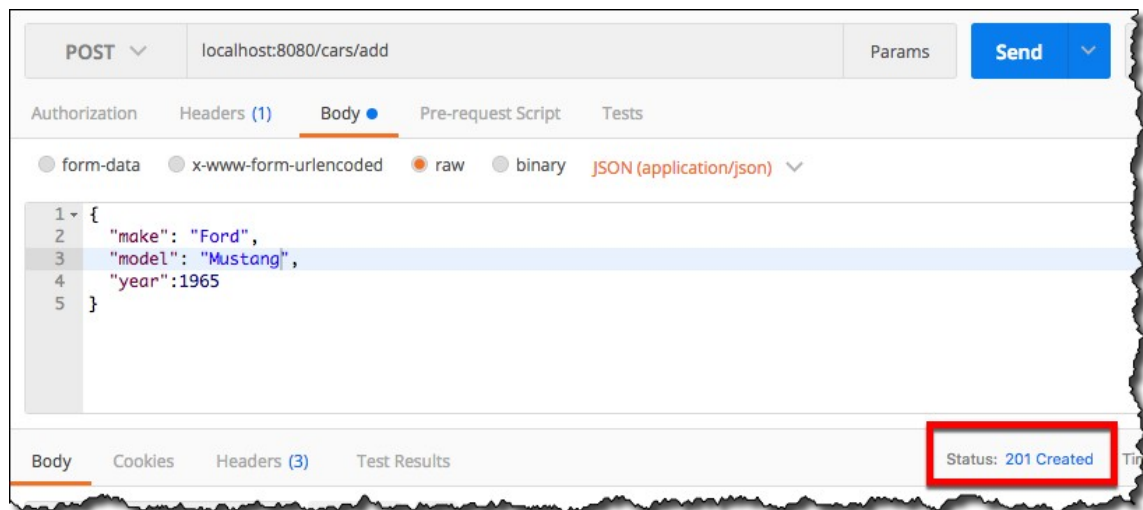
- \_\_ 10. Click **Body** and select **raw**.
- \_\_ 11. Add a car to the database with the following parameters.

\_\_12. Enter is the body the following:

```
{
  "year":1965,
  "make":"Ford",
  "model":"Mustang"
}
```

\_\_13. Click **Send**.

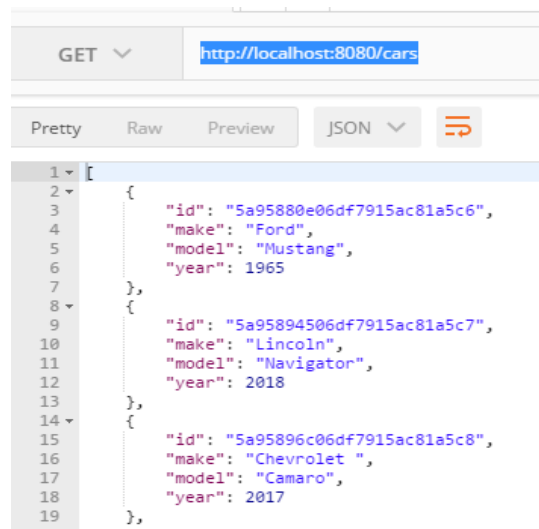
Notice the returned status is **201 Created**.



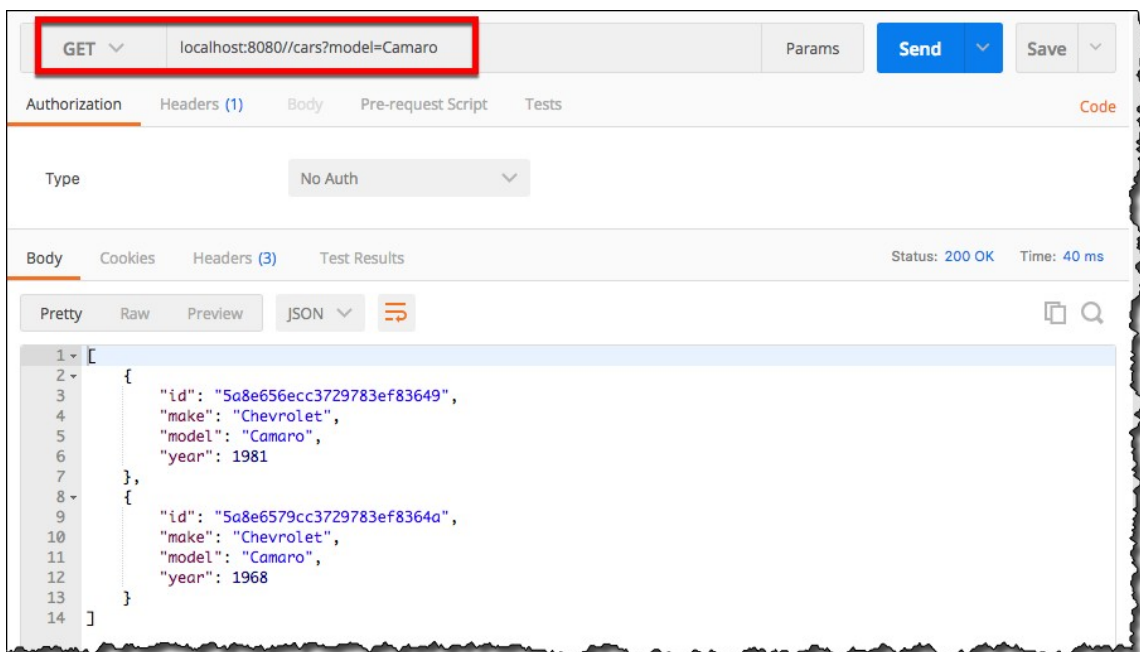
\_\_14. Use Postman to add multiple cars to the database, be creative.

Now we will use Postman to retrieve cars from the database.

\_\_15. Select **GET**, enter the URL **http://localhost:8080/cars** and click **Send** to retrieve all cars from the database.



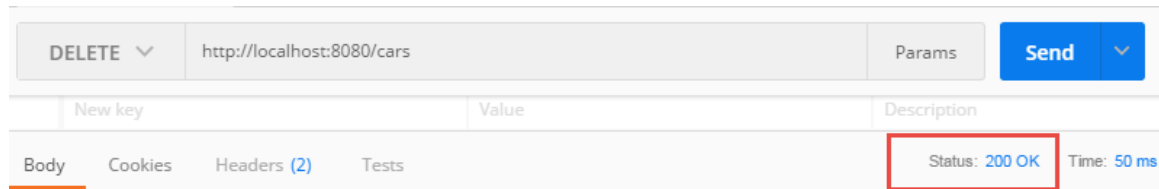
\_\_16. Use Postman to retrieve cars from the database. Example we are using **GET** and setting the URL to **http://localhost:8080/cars?model=Camaro** (Modify the query string to search for a model car that you entered in the database.)



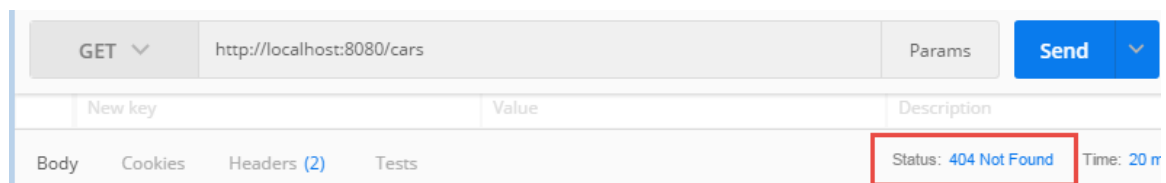
\_\_17. Try variations on the URL to retrieve cars with different parameters.

\_\_18. Use Postman to delete all the Cars in the database with the following URL:

DELETE URI: `http://localhost:8080/cars`



\_\_19. Use Postman to call the service and verify that the Cars have all been deleted.



\_\_20. In the <mongo> command window, press **Ctrl-C** (you may need to repeat this command again to terminate the process).

\_\_21. In the MONGO SHELL window, execute this command:

**exit**

\_\_22. In the MONGOD command window, press Ctrl-C (you may need to repeat this command again to terminate the process).

\_\_23. Close all.

## Part 6 - Review

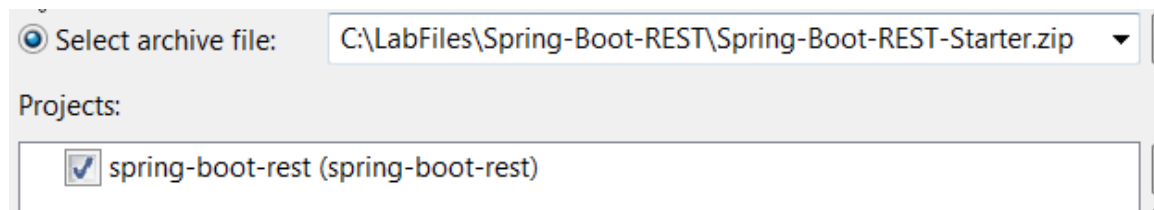
In this lab we built a Microservice that used a RESTful interface to manage Car objects and store them in a MongoDB database.

## Lab 7 - Create a RESTful API with Spring Boot

In this lab you will use Spring Boot to implement a RESTful API for a repository of customers, similar to what you'd need for an online store.

### Part 1 - Import the Starter Project

1. From Eclipse's main menu, click **File** → **Import** and then select **General** → **Existing Projects into Workspace**, and then click **Next**.
2. Click the radio button for **Select archive file**:
3. Click the **Browse** button and navigate to select **C:\LabFiles\Spring-Boot-REST\Spring-Boot-REST-Starter.zip** and then click **Open**.
4. On the **Import** dialog, leave the defaults as-is and click **Finish**.



### Part 2 - Examine the Starter Project

The starter project implements a JPA-based Repository for Customer domain objects using the Spring Data components. If you've just finished the Spring JPA lab, you'll already be familiar with the project.

There are a few items already setup in the starter project that you should take note of for your own projects.

- 'pom.xml' contains two dependencies that are particularly important to this project:

```
<dependency>
  <groupId>org.hsqldb</groupId>
  <artifactId>hsqldb</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

- These dependencies pull in the components required for Spring Boot to implement the Spring Data framework, and also to use the embedded HSQLDB instance for testing.

- In 'src/main/resources', there are two files, 'data.sql' and 'schema.sql', which establish the database schema and initial data load for the embedded database that we'll use for testing. These files end up being assembled to the root of the classpath, where Spring Boot can load them at startup.
- Also in 'src/main/resources', there is a file called 'application.properties'. This file contains one line:  
  
`spring.jpa.hibernate.ddl-auto=false`
- The line disables Hibernate's automatic schema generation, which would overwrite our test data if we left it enabled.
- The starter project contains a pair of domain classes that we will use as a starting point for our repository.
- The starter project also contains a rudimentary user interface for our demonstration program.

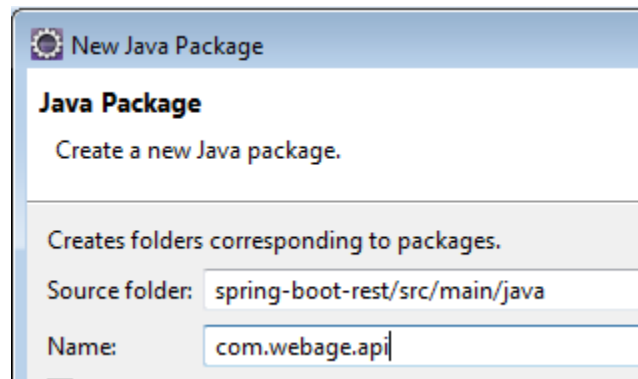
1. Right click **spring-boot-rest** and select **Maven → Update Project**.
2. Make sure **spring-boot-rest** is selected and click OK.
3. Right click on the **spring-boot-rest** project and select **Run as → Maven install** to Run a Maven install and ensure that there are no errors. You may need to run the Maven Install two times to get the Build Success message.

### Part 3 - Create the Customer API Class

Although we're running under the Spring Boot environment, the basic technology for creating RESTful services is contained in the Spring MVC framework. To create an API under this framework, we will create one or more classes that have methods to serve out responses to HTTP requests. In this particular case, there is no real business logic involved in the API, so the methods will essentially delegate all their work to the Repository class. As such, the methods are short, and a full implementation for the four core HTTP methods will be small enough to fit into one API class. We'll implement GET, POST, and PUT methods (we'll assume that we never delete a customer).

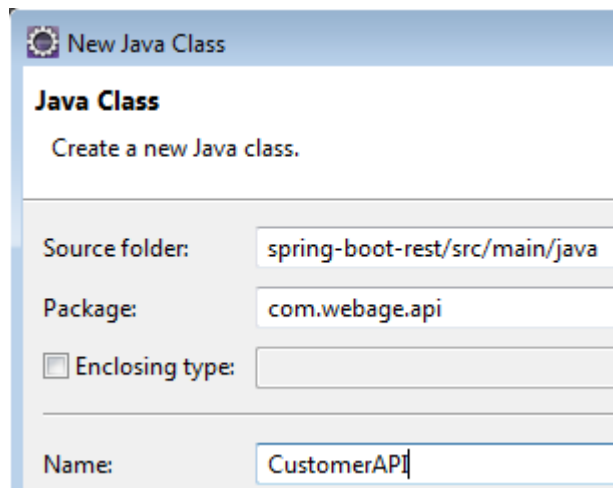
1. In the **Project Explorer**, locate the 'spring-boot-rest/src/main/java' node.
2. Right-click on 'src/main/java' and select **New → Package**.

\_\_\_3. Enter 'com.webage.api' as the package name and then click **Finish**.



\_\_\_4. Right-click on the newly-created package and select **New** → **Class**.

\_\_\_5. Enter 'CustomerAPI' as the class name and then click **Finish**.



\_\_\_6. We need to flag this class as a resource controller and assign a URL path for it. To do that, add annotations to the class as shown below:

```
@RestController
@RequestMapping("/customers")
public class CustomerAPI {
```

These annotations will make this resource controller serve out the "/customers" resource path.

\_\_7. Inside the body of the class, add an '@Autowired' reference to the 'CustomersRepository' implementation, as shown below:

```
public class CustomerAPI {  
  
    @Autowired  
    CustomersRepository repo;  
  
}
```

## Part 4 - Implement a GET method

First, let's implement the 'GET' method against the '/customers' resource path. This should return all the customers that are in the repository.

\_\_1. In the body of the 'CustomerAPI' class, add the following method. It will use the repository proxy to retrieve an 'Iterable' object that iterates through all the customers.

```
public Iterable<Customer> getAll() {  
    return repo.findAll();  
}
```

\_\_2. We want this method to respond to a 'GET' request on the '/customers' path. Since the API class is already annotated with the required path, all we need to do is flag this class as a 'GET' method. To do so, add the '@GetMapping' annotation to the method as shown below:

```
@GetMapping  
public Iterable<Customer> getAll() {  
  
}
```

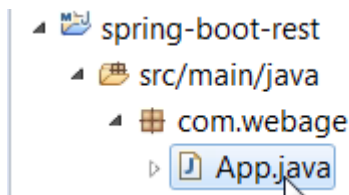
\_\_3. Organize the imports by pressing **Ctrl-Shift-O**.

\_\_4. Save the file.

## Part 5 - Test the GET Method

\_\_1. Using the same techniques as in previous labs, run a 'Maven install'.

\_\_2. Run the 'App' class as a **Java Application**.



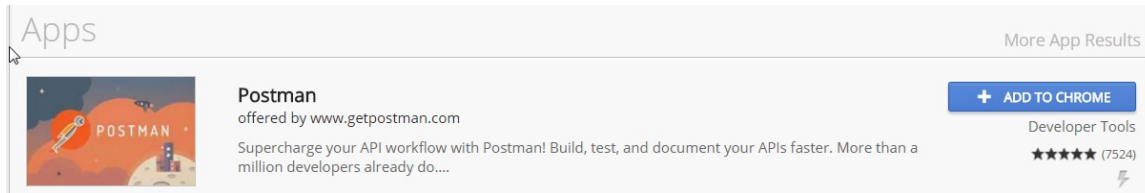
The system should start up without errors.



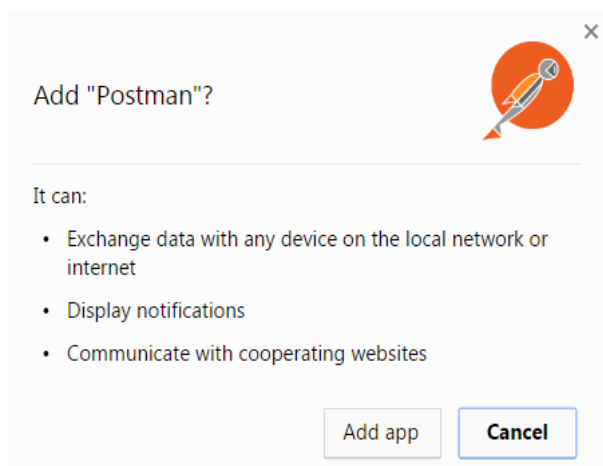
\_\_3. Open Chrome and find the Postman App.

<https://chrome.google.com/webstore/search/postman>

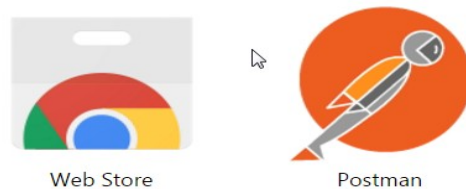
\_\_4. Select the **Postman** under **Apps** and click **ADD TO CHROME**.



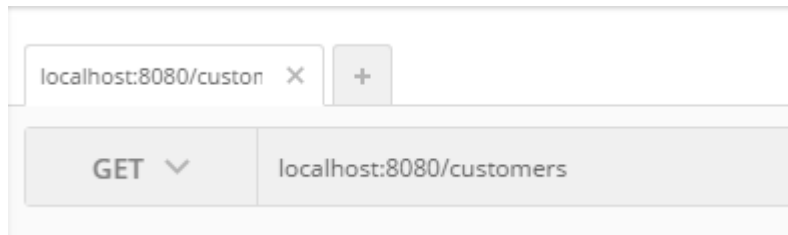
\_\_5. Click **Add App**.



\_\_6. A new tab showing <chrome://apps/> will open, click **Postman**.

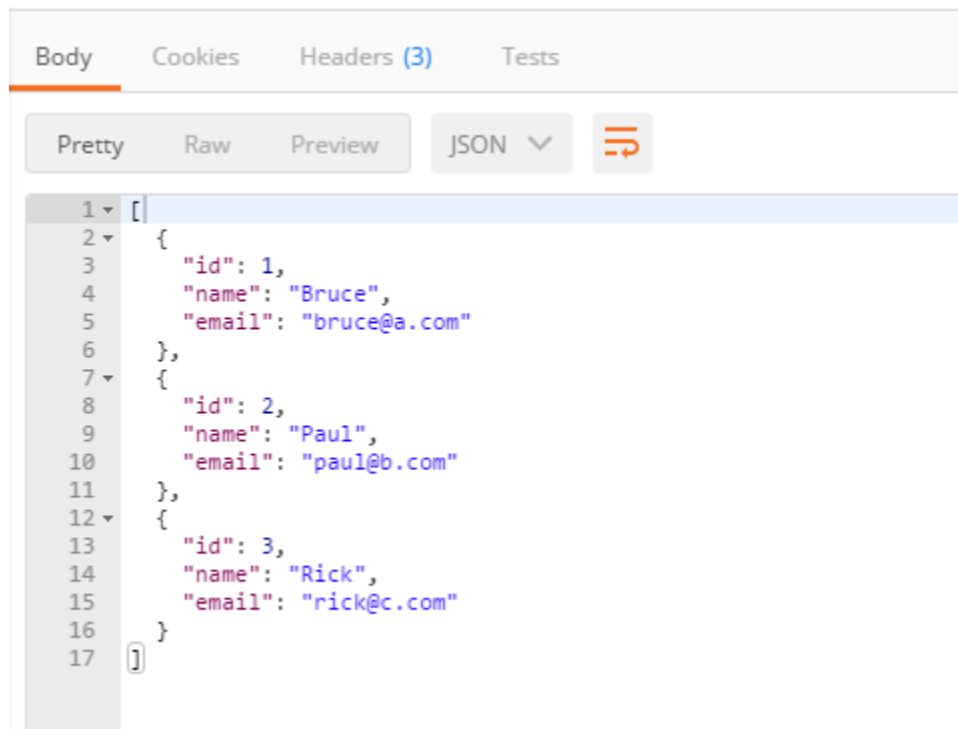


\_\_\_7. Select the a 'GET' request using the drop-down box, and enter 'localhost:8080/customers' as the URL:



\_\_\_8. Click the **Send** button.

The result should look like below (you may need to select the 'Body' tab to see the result):



So far, so good! Leave **Postman** open - we'll be using it for the next test.

## Part 6 - Lookup a Specific Customer by ID

Looking up a particular customer could be modeled as a "GET" request to a path like "/customers/2", where '2' is the customer ID.

\_\_1. Back in eclipse, in the body of the 'CustomerAPI' class, add the following method:

```
public Customer getCustomerById( long id) {  
    return repo.findOne(id);  
}
```

The method takes the id as a parameter and looks up the corresponding customer.

\_\_2. Annotate the method with `@GetMapping` as follows:

```
@GetMapping("/{customerId}")
```

Notice that we've put a path variable in the path, '{customerId}'. We need that path variable to be fed into our lookup method.

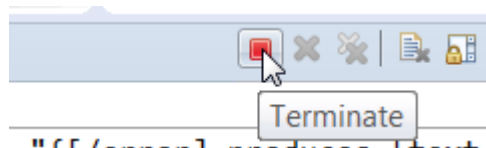
\_\_3. Annotate the 'long id' parameter as follows:

```
public Customer getCustomerById(@PathVariable("customerId") long id) {
```

\_\_4. Organize imports.

\_\_5. Save the file.

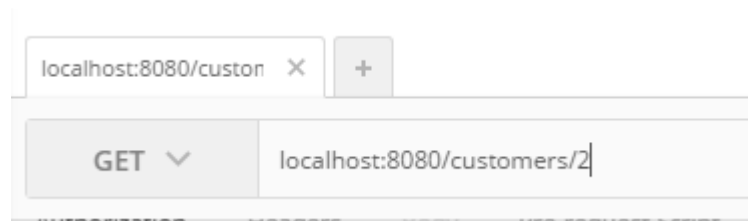
\_\_6. If you still have 'App.java' running from the previous test, stop it using the red stop button in the console window.



\_\_7. Run 'Maven install'.

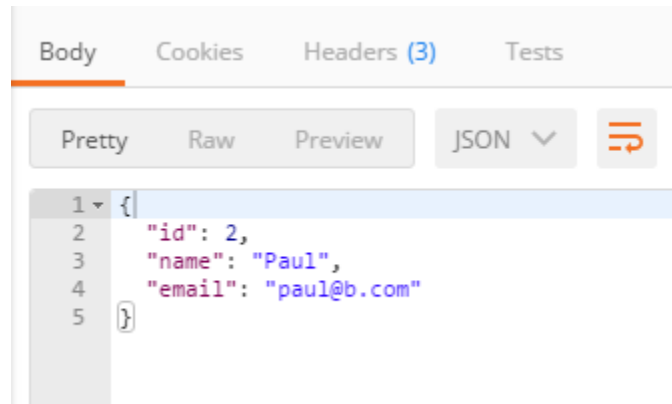
\_\_8. Run 'App.java'.

\_\_9. In the **Postman** tool, add an identifier to the end of the URL:



\_\_10. Click **Send**.

We should get a single result:



\_\_11. Once again, leave **Postman** open, but stop 'App.java'.

## Part 7 - Add a POST Method Handler

The POST method is intended to add an entity to a collection of entities. For instance, adding a new customer to the set of customers. This action is accurately modeled as a "POST" to the "/customers" path.

\_\_1. In the 'CustomerAPI' class, add the following method.

```
@PostMapping
public ResponseEntity<?> addCustomer(@RequestBody Customer newCustomer,
                                     HttpRequest request, UriComponentsBuilder uri) {
    if (newCustomer.getId() != 0
        || newCustomer.getName() == null
        || newCustomer.getEmail() == null) { // Reject - we'll assign the
customer id
        return ResponseEntity.badRequest().build();
    }
    newCustomer=repo.save(newCustomer);
    URI location=ServletUriComponentsBuilder.fromCurrentRequest()
        .path("/{id}").buildAndExpand(newCustomer.getId()).toUri();
    ResponseEntity<?> response=ResponseEntity.created(location).build();
    return response;
}
```

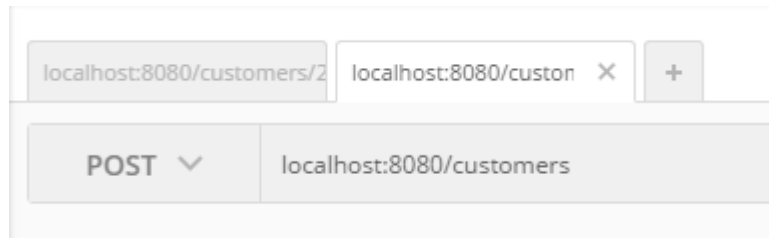
\_\_2. Organize imports.

\_\_3. Save the file.

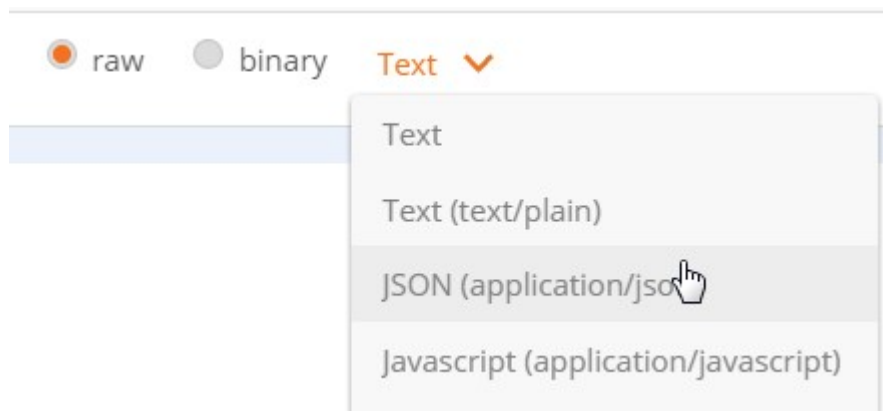
\_\_4. Run 'Maven install'.

\_\_5. Run 'App.java'.

- \_\_6. In the **Postman** tool, open a new tab by clicking on the '+' icon in the tab headers.
- \_\_7. Click the 'method' drop-down box and select 'POST'.
- \_\_8. Enter 'localhost:8080/customers' as the request URL.



- \_\_9. Select the 'Body' tab and then click the radio button for 'raw'.
- \_\_10. Expand the drop-down box and select 'JSON(application/json)'

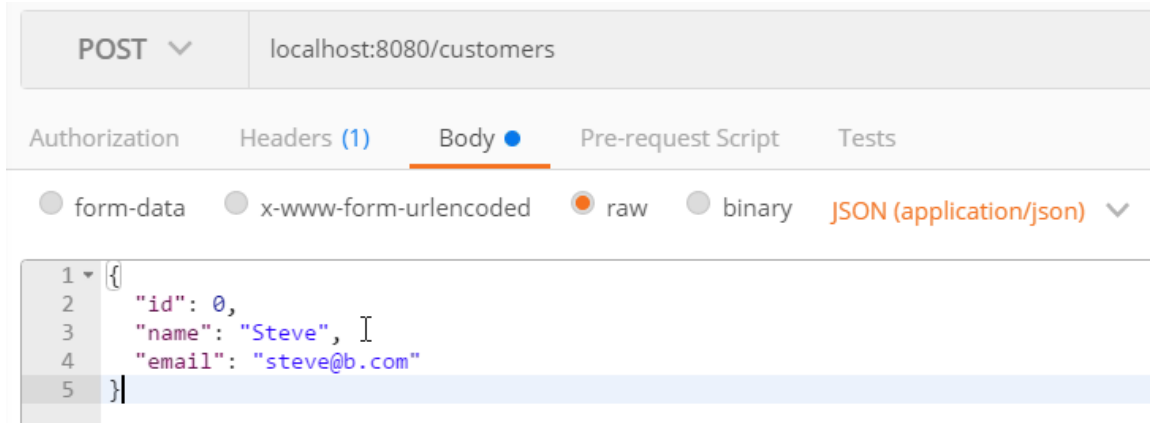


- \_\_11. Enter the following new customer object in the body text area (hint: you might find it convenient to copy and edit an object from the response to the 'GET' request that we issued earlier - it's in the other tab in **Postman**).

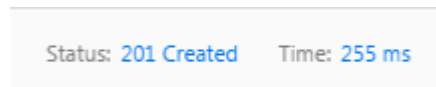
```
1 {  
2   "id": 0,  
3   "name": "Steve",  
4   "email": "steve@b.com"  
5 }
```

Note: Make sure you set the 'id' value to 0.

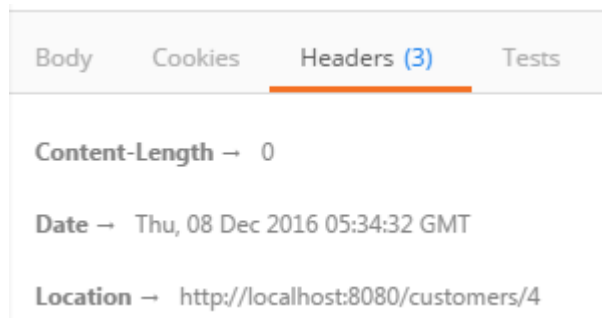
\_\_12. When it looks like below click **Send**.



The response should show a '201 Created' status. Look at the bottom section.



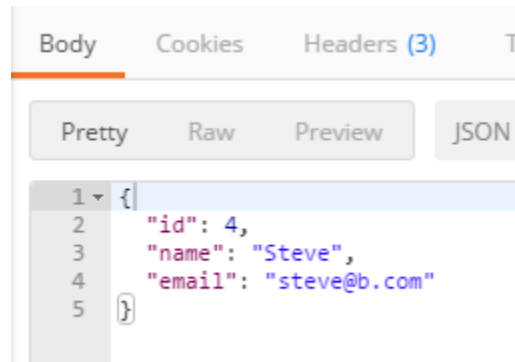
\_\_13. Click on the **Headers** tab in the response area [bottom section]. You should see the 'Location' header:



The location header gives us the URL for the newly created customer object. Notice that the repository assigned the next customer id for us.

\_\_14. Copy the contents of the 'Location' header into the URL area and issue a GET request.

You should see the new Customer object similar to the following:



\_\_15. Leave **Postman** open, but stop 'App.java'

## Part 8 - Add a PUT Method Handler

The PUT method models an update to a particular customer. For instance to change the customer's email for customer id 4, we would PUT an updated object to '/customers/4'.

\_\_1. Add the following method to the 'CustomerAPI' class.

```
@PostMapping("/{customerId}")
public ResponseEntity<?> putCustomer(@RequestBody Customer newCustomer,
    @PathVariable("customerId") long customerId) {
    if (newCustomer.getId() != customerId
        || newCustomer.getName() == null
        || newCustomer.getEmail() == null) {
        return ResponseEntity.badRequest().build();
    }
    newCustomer = repo.save(newCustomer);
    return ResponseEntity.ok().build();
}
```

\_\_2. Organize the imports by pressing **Ctrl-Shift-O**.

\_\_3. Save the file.

Notice that we're verifying the customer id in the supplied object matches the id that's in the path variable. If there's a difference, we return a 'badRequest' response.

\_\_4. Run 'Maven install'.

\_\_5. Run 'App.java'.

\_\_6. In the **Postman** tool, open a new tab by clicking on the '+' icon in the tab headers.

\_\_7. Click the 'method' drop-down box and select 'PUT'.

\_\_8. Enter 'localhost:8080/customers/4' as the request URL.

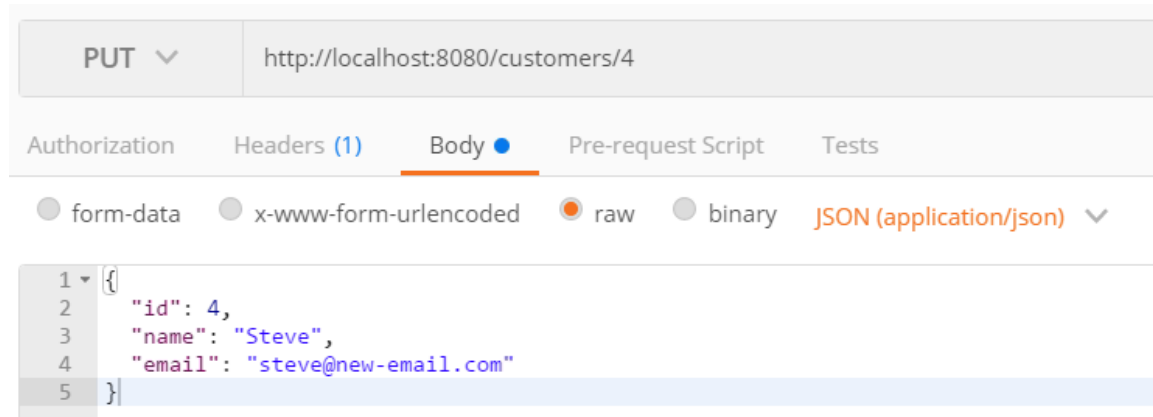
\_\_9. Select the 'Body' tab and then click the radio button for 'raw'.

\_\_10. Expand the drop-down box and select 'JSON(application/json)'

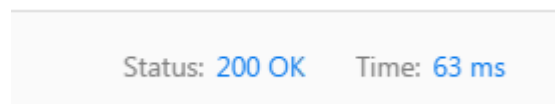
\_\_11. Enter the following new customer object in the body text area.

```
{
  "id": 4,
  "name": "Steve",
  "email": "steve@new-email.com"
}
```

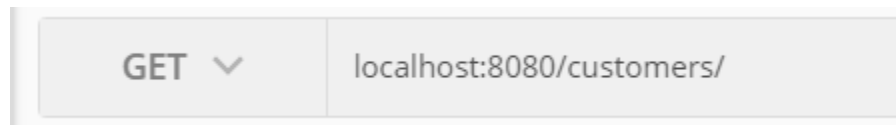
\_\_12. When looks like below click **Send**.



You should see a '200 OK' response.

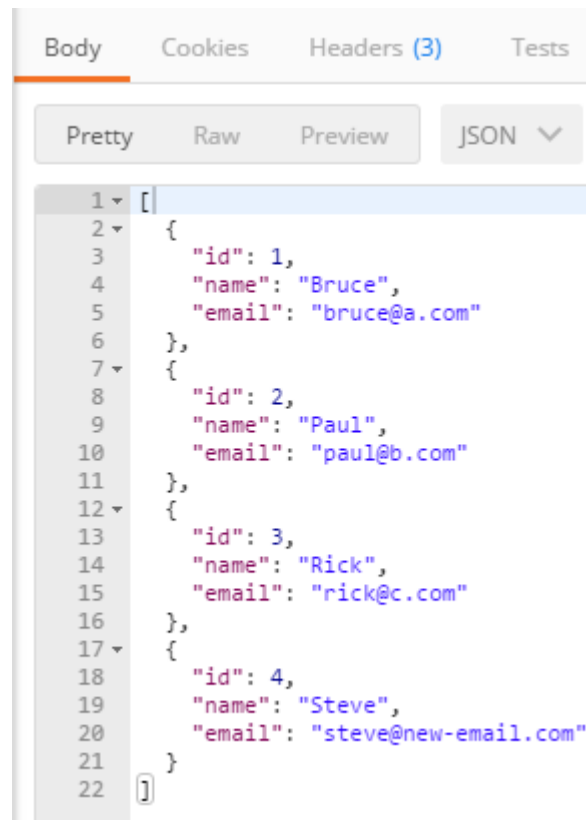


\_\_13. In a different tab in **Postman**, do a 'GET' request to '/customers'





Notice that the record for customer id 4 has been updated with the new email address we supplied.



```
1 [
2   {
3     "id": 1,
4     "name": "Bruce",
5     "email": "bruce@a.com"
6   },
7   {
8     "id": 2,
9     "name": "Paul",
10    "email": "paul@b.com"
11  },
12  {
13    "id": 3,
14    "name": "Rick",
15    "email": "rick@c.com"
16  },
17  {
18    "id": 4,
19    "name": "Steve",
20    "email": "steve@new-email.com"
21  }
22 ]
```

- \_\_14. Close Postman.
- \_\_15. Back in eclipse, close all open files.
- \_\_16. Stop 'App.java' by clicking on the red stop button.

## Part 9 - Review

In this lab, we used Spring MVC to implement a CRUD service that stores Customer objects. Thanks to Spring's automatic mapping to JSON, we had to do very little coding to realize the API.

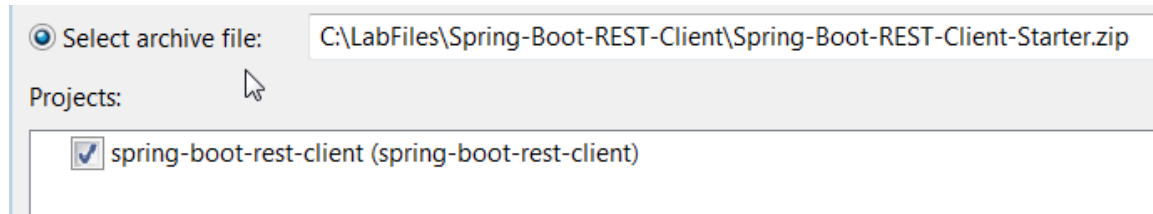
## Lab 8 - Create a RESTful Client with Spring Boot

In this lab you will use Spring's RestTemplate class to make a call to a RESTful API to retrieve a list of Customers.

Note: This lab requires that the 'spring-boot-rest' application is running. If necessary, load the project from the solutions and run 'App.java'.

### Part 1 - Import the Starter Project

1. From Eclipse's main menu, click **File** → **Import** and then select **General** → **Existing Projects into Workspace**, and then click **Next**.
2. Click the radio button for **Select archive file**:
3. Click the **Browse** button and navigate to select **C:\LabFiles\Spring-Boot-REST-Client\Spring-Boot-REST-Client-Starter.zip** and then click **Open**.
4. On the **Import** dialog, leave the defaults as-is and click **Finish**.



### Part 2 - Examine the Starter Project

The starter project implements a web page to display a list of Customer objects. There is a CustomerDAO interface provided, but no implementation of that interface.

There are a few items already setup in the starter project that you should take note of for your own projects

- Also in 'src/main/resources', there is a file called 'application.yml'. This file contains the following content:

```
---
server:
  port: 8081
```

- The line disables sets up the embedded web server to run on port 8081. We need to configure this for our test, because the API that we'll be hitting is already running on port 8080.
- The starter project contains a domain class that we will use as a starting point for our repository.

- The starter project also contains a rudimentary user interface for our demonstration program.

1. Right click **spring-boot-rest-client** and select **Maven → Update Project**.
2. Make sure **spring-boot-rest-client** is selected and click OK.
3. Right click on the **spring-boot-rest-client** project and select **Run as → Maven install** to Run a Maven install and ensure that there are no errors. You may need to run the Maven Install two times to get the Build Success message.

### Part 3 - Complete the CustomerDAO

The repository class 'CustomerDAO' implements the CustomersDAO interface, which is called by the user interface's controller object. In the starter project, the implementation is incomplete; it just returns an empty ArrayList of Customers. We'll change that to get the list of customers from a RESTful API.

1. In the **Project Explorer**, locate the 'APIClientCustomersDAO' class in the 'spring-boot-rest-client/src/main/java/com.webage.dao' package. Double-click on the file to open it.
2. Add the following line inside the class to provide the connection URL as an instance variable:

```
String customersAPIbase="http://localhost:8080/customers";
```

3. Look for the 'Insert code here...' comment inside the 'getAllCustomers' method.
4. Edit the method to look like this:

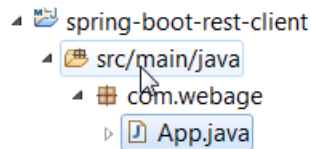
```
@Override
public Collection<Customer> getAllCustomers() {
    // Construct a GET request to the CustomersAPI base url
    // Insert code here..
    RestTemplate template=new RestTemplate();
    Customer[] customers=template.getForObject(customersAPIbase,
Customer[].class);
    return Arrays.asList(customers);
}
```

The method creates an instance of RestTemplate, and then calls the 'getForObject(...)' method to retrieve and convert a JSON response from the server.

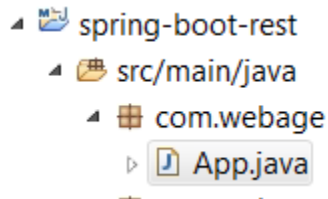
5. Organize imports by pressing **Ctrl-Shift-O**. Select **java.util.Arrays**.
6. Save the file.

\_\_7. Run a 'Maven install'.

\_\_8. Run the 'App.java' to startup the server on the **spring-boot-rest-client** project.



\_\_9. Run the 'App.java' to startup the server on the **spring-boot-rest** project. [Previous Lab].



\_\_10. Open a Web Browser and enter 'http://localhost:8081/browseCustomers' as the url.

You should see a response similar to:

## Browse Purchases

[Add Purchase](#)

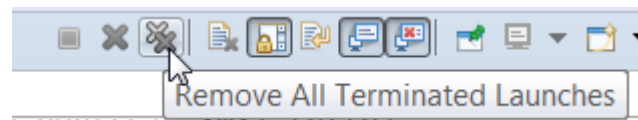
Customer Id	Customer Name	Email
1	Bruce	bruce@a.com
2	Paul	paul@b.com
3	Rick	rick@c.com

[Back to Main Menu](#)

\_\_11. Close the browser.

\_\_12. Stop 'App.java' by clicking on the red stop button.

\_\_13. Click **Remove All Terminated Launches**.



\_\_14. Repeat the previous 2 steps as many times as needed until your console is clean.

\_\_15. Close all open files.

## **Part 4 - Review**

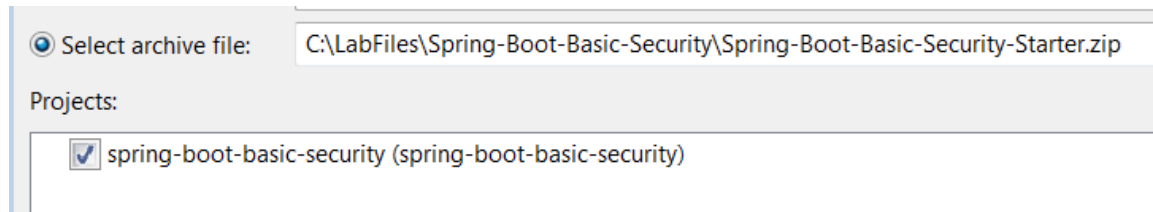
In this lab, we used Spring's RestTemplate object to quickly implement a client to a RESTful service.

## Lab 9 - Enable Basic Security

In this lab we will enable basic security on a Spring Boot web application.

### Part 1 - Import the Starter Project

1. From Eclipse's main menu, click **File** → **Import** and then select **General** → **Existing Projects into Workspace**, and then click **Next**.
2. Click the radio button for **Select archive file**:
3. Click the **Browse** button and navigate to select **C:\LabFiles\Spring-Boot-Basic-Security\Spring-Boot-Basic-Security-Starter.zip** and then click **Open**.

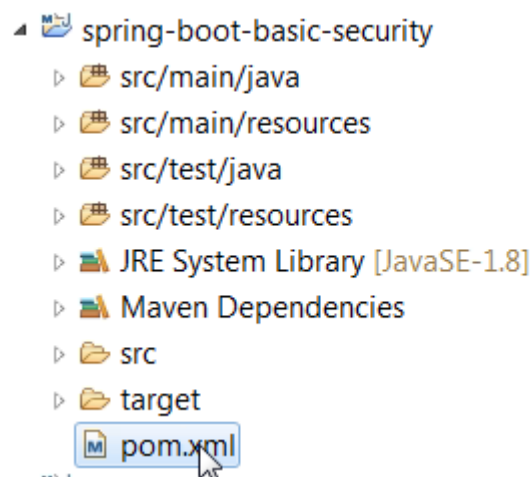


4. On the **Import** dialog, leave the defaults as-is and click **Finish**.

### Part 2 - Enable Security

There actually isn't much work required to enable security. Spring Boot looks in the classpath for various modules when it starts up, and enables them if they are present. So all we need to do to enable basic security is to put Spring Security on the classpath. The Spring Boot project provides a starter package that does exactly that...

1. In the **Project Explorer**, locate 'pom.xml' and double-click on it to open the file. When the editor opens, click on the 'pom.xml' tab at the bottom of the editor panel to view the plain xml.



\_\_2. Add the following dependency element just before the closing `</dependencies>` tag:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

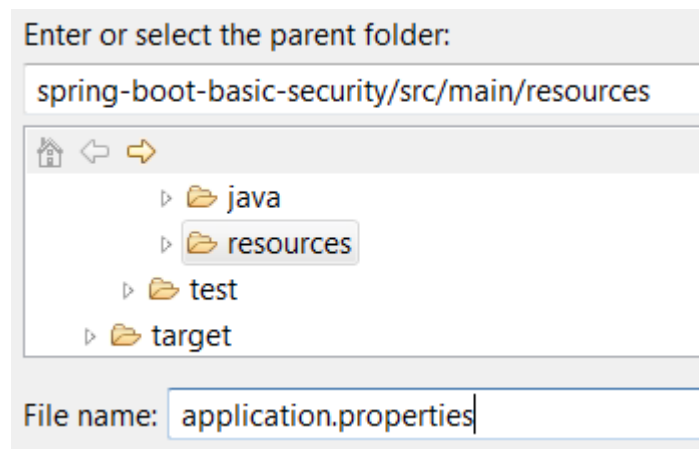
\_\_3. Save and close the file.

### Part 3 - Configure the User Password

\_\_1. In the **Project Explorer**, locate the node for 'src/main/resources'.

\_\_2. Right-click on 'src/main/resources' and then select **New-> Other**, and select **General** → **File** in the dialog and click **Next**.

\_\_3. Enter 'application.properties' as the filename and click **Finish**.



\_\_4. In the new file, enter the following line:

```
security.user.password=Pa$$w0rd
```

\_\_5. Save and close the file.

### Part 4 - Build and Test

Using the same technique as in previous labs, you will run "Maven install" and then run "App.java" as a Java Application.

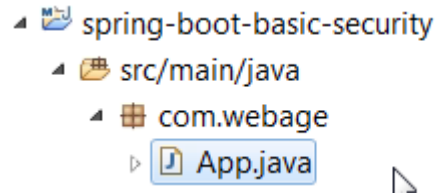
\_\_1. Right click **spring-boot-basic-security** and select **Maven** → **Update Project**.

\_\_2. Make sure **spring-boot-basic-security** is selected and click OK.

\_\_3. Right click on the **spring-boot-basic-security** project and select **Run as** → **Maven install** to Run a Maven install and ensure that there are no errors. You may need to run

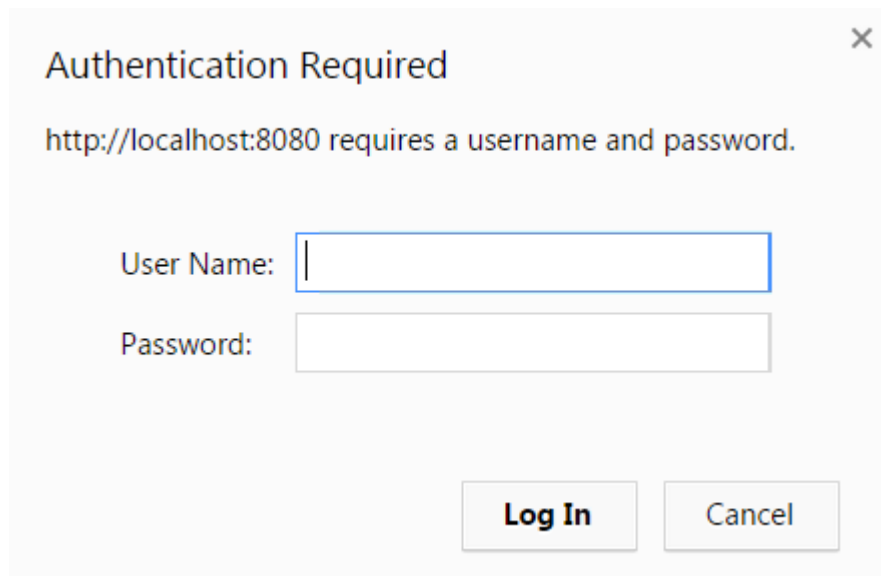
the Maven Install two times to get the Build Success message.

\_\_\_4. Run the 'App.java' to startup the server on the **spring-boot-basic-security**.



\_\_\_5. Open a browser and navigate to 'localhost:8080'.

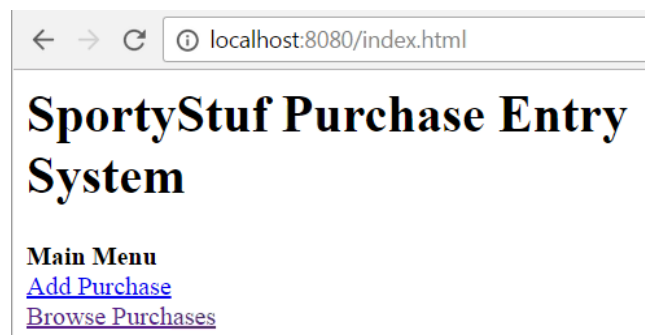
\_\_\_6. You will be prompted for a user name and password.



\_\_\_7. Enter 'user' as the username and 'Pa\$\$w0rd' as the password and then click Login.

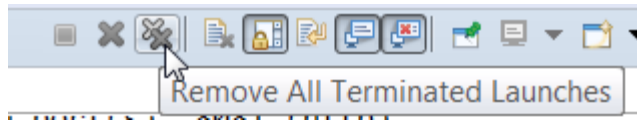
\_\_\_8. Do not save the password.

\_\_\_9. Once you've entered your password, you will have full access to the application.





- \_\_10. Close the browser.
- \_\_11. Stop 'App.java' by clicking on the red stop button.
- \_\_12. Click **Remove All Terminated Launches**.



- \_\_13. Close all open files.

## Part 5 - Review

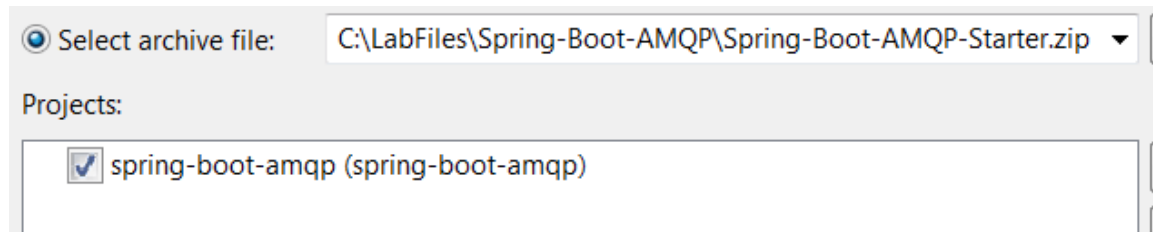
This was a very short lab demonstrating how simple it is to add basic security to an application that is built using Spring Boot.

## Lab 10 - Use AMQP Messaging with Spring Boot

In this lab you will complete a Spring Boot application that sends and receives messages through a RabbitMQ instance.

### Part 1 - Import the Starter Project

- \_\_\_ 1. From Eclipse's main menu, click **File** → **Import** and then select **General** → **Existing Projects into Workspace**, and then click **Next**.
- \_\_\_ 2. Click the radio button for **Select archive file**:
- \_\_\_ 3. Click the **Browse** button and navigate to select **C:\LabFiles\Spring-Boot-AMQP\Spring-Boot-AMQP-Starter.zip** and then click **Open**.



- \_\_\_ 4. On the **Import** dialog, leave the defaults as-is and click **Finish**.

### Part 2 - Examine the Starter Project

The starter project implements a simple HTML form that accepts inputs that simulate an Order in an online-store system. Most but not all of the components are in place to send that order into an AMQP message queue. We'll fill in the rest of the components in this lab. We'll also add a component that listens on the same message queue and prints out the orders that are received.

Note: In normal application usage, we would have one application sending the messages, and a completely different application receiving the messages. We're only combining the two functions in one application here to simplify the lab setup a little.

There are a few items already setup in the starter project that you should take note of for your own projects

- 'pom.xml' contains a dependency that are particularly important to this project:

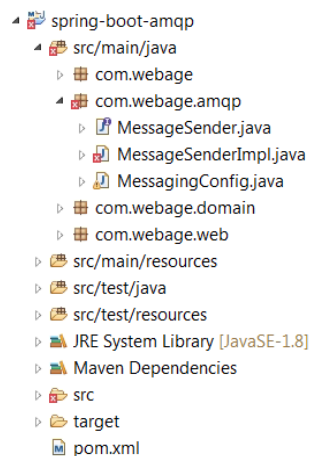
```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

- This dependency pulls in the AMQP and RabbitMQ support.
- In 'src/main/resources/templates', there are HTML files that are used to display the data entry and success forms.
- The basic components of the Spring Boot application are already present. There is an 'App.java' file that contains a 'Main' method that can be used to start the system.
- The 'com.webage.amqp' package contains starter point files for the messaging implementation.
- The 'com.webage.web' package contains a request handler class that receives the form submittal and calls an implementation of 'com.webage.amqp.MessageSender' that is autowired by Spring.
- At the root of 'src/main/resources', there is a YAML file, 'application.yml' that contains the connection details (host, port, userid, user secret) for connecting to the RabbitMQ instance.

### Part 3 - Complete the MessageConfig

Spring Boot handles the vast majority of the setup automatically, as soon as it finds the AMQP libraries in the classpath. However, there is one piece of setup that we need to provide: We need to have a configuration object that will enable the messaging queue to be created on-demand.

1. In the **Project Explorer**, locate the 'com.webage.amqp' package node.



2. Inside the 'com.webage.amqp' package, locate the 'MessagingConfig.java' file. Double-click on the file to open it. You will see that the class is empty.

\_\_3. Add the following declaration to the body of the class:

```
@Bean
public Queue myQueue() {
    return new Queue("myqueue1");
}
```

This declaration declares a desired message queue. Spring AMQP will use this definition to create the queue the first time it's used.

\_\_4. Save and close the file.

## Part 4 - Complete the MessageSenderImpl

\_\_1. In the 'com.webage.amqp' package, locate the 'MessageSenderImpl' class, and then double-click on the class to open it.

\_\_2. The class should look something like this:

```
@Component
public class MessageSenderImpl implements MessageSender {

    @Autowired
    private AmqpTemplate amqpTemplate=null;

}
```

The class is requesting injection of an 'AmqpTemplate' object. This object is defined automatically by Spring Boot when we include the starter dependency.

\_\_3. Add the following method into the body of the class:

```
@Override
public void sendOrderMessage(Order toSend) {
    System.out.println("Sending message with order - " + toSend);
    amqpTemplate.convertAndSend("myqueue1", toSend);
}
```

This is the method that the web controller component will call to actually send the message.

\_\_4. Save the file. Bite that the error is gone.

\_\_5. Right click **spring-boot-amqp** and select **Maven → Update Project**.

\_\_6. Make sure **spring-boot-amqp** is selected and click OK.

\_\_7. Right click on the **spring-boot-amqp** project and select **Run as → Maven install** to Run a Maven install and ensure that there are no errors. You may need to run the Maven Install two times to get the Build Success message.

## Part 5 - Add a Message Listener

In order to simplify the lab a little bit, we're going to create a message listener in the same project as the message sender. This structure is probably not one that you would use in production, but it's not entirely out of the question, either.

- \_\_\_ 1. Create a new class called 'MyListener' in the 'com.webage.amqp' package.
- \_\_\_ 2. Edit the class so the body of it looks like:

```
@Component
public class MyListener {

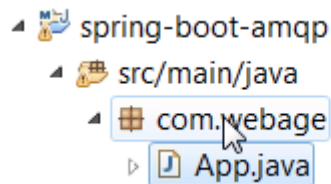
    @RabbitListener(queues="myqueue1")
    public void onMessage(Order order) {
        System.out.println("Message received - Order Details:" + order);
    }
}
```

- \_\_\_ 3. Organize imports by pressing **Ctrl-Shift-O**. Select **com.webage.domain.Order**.
- \_\_\_ 4. Save the file.

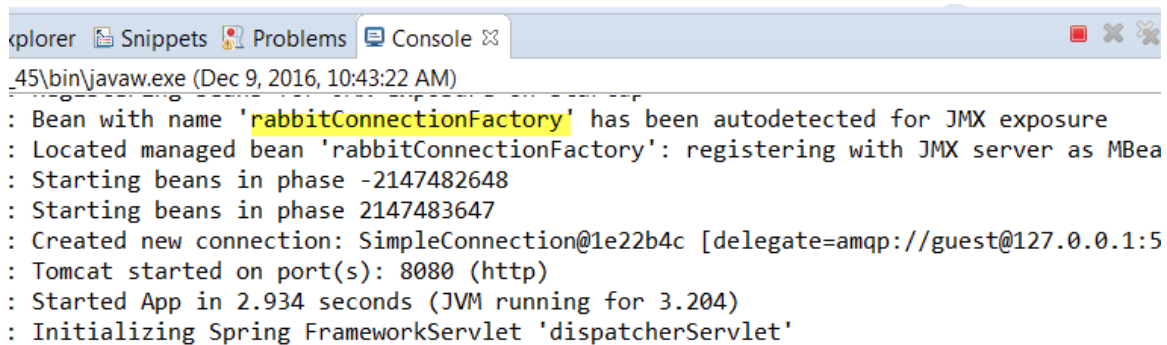
The '@RabbitListener' annotation will establish this method as a listener on the designated queue. Spring Boot will setup an additional thread to monitor the queue and then call this method when a message arrives. Here, the message handler method simply prints out the received message to the console.

## Part 6 - Test

- \_\_\_ 1. Run 'Maven Install'.
- \_\_\_ 2. Run the 'App.java' class as a Java application.

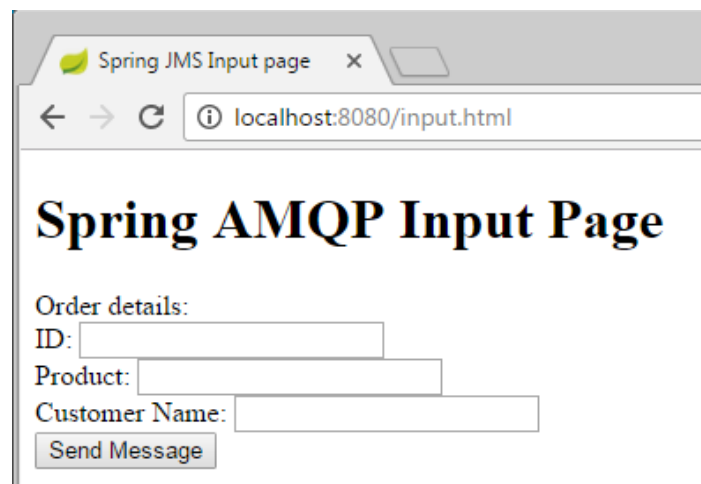


\_\_3. Notice that rabbit has been called.



```
explorer Snippets Problems Console
_45\bin\javaw.exe (Dec 9, 2016, 10:43:22 AM)
: Bean with name 'rabbitConnectionFactory' has been autodetected for JMX exposure
: Located managed bean 'rabbitConnectionFactory': registering with JMX server as MBea
: Starting beans in phase -2147482648
: Starting beans in phase 2147483647
: Created new connection: SimpleConnection@1e22b4c [delegate=amqp://guest@127.0.0.1:5
: Tomcat started on port(s): 8080 (http)
: Started App in 2.934 seconds (JVM running for 3.204)
: Initializing Spring FrameworkServlet 'dispatcherServlet'
```

\_\_4. Open a web browser and navigate to 'localhost:8080/input.html'.

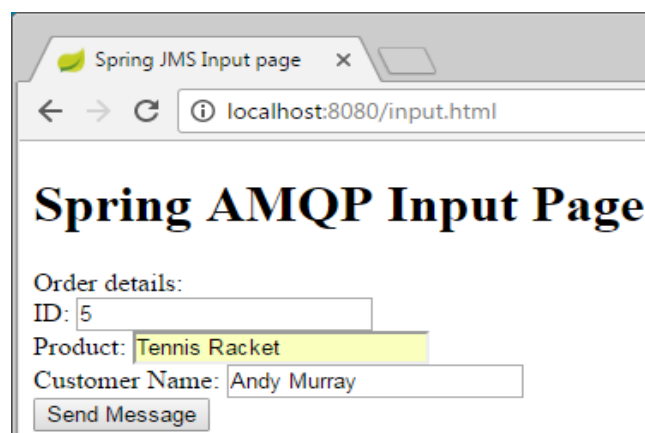


\_\_5. Enter in the following order details:

ID: 5

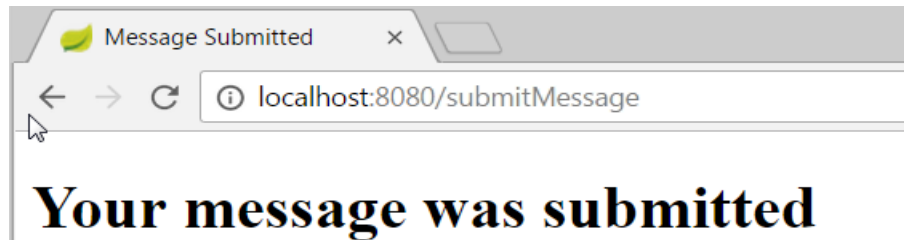
Product: Tennis Racket

Customer Name: Andy Murray



\_\_6. Click on **Send Message**.

\_\_7. You will see this message:



\_\_8. Look in the system console: You should see output that suggest the message was sent and received.

```
2016-12-09 03:17:28.223 INFO 3720 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : F
2016-12-09 03:17:28.254 INFO 3720 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : F
getIndex was called
Order submitted to RequestHandlerServlet: Order #:5, Product:Tennis Racket, Customer:Andy Murray
Sending message with order - Order #:5, Product:Tennis Racket, Customer:Andy Murray
Message received - Order Details:Order #:5, Product:Tennis Racket, Customer:Andy Murray
```

\_\_9. Close the browser.

\_\_10. Stop 'App.java' by clicking on the red stop button.

\_\_11. Click **Remove All Terminated Launches**.

\_\_12. Close all open files.

## Part 7 - Review

In this lab, we used Spring AMQP to send message into the messaging server. As we saw, Spring Boot handles the vast majority of the configuration automatically, leaving only a minimal amount of work for the programmers to add messaging capability to an application.

## Lab 11 - Use Netflix Eureka for Service Discovery

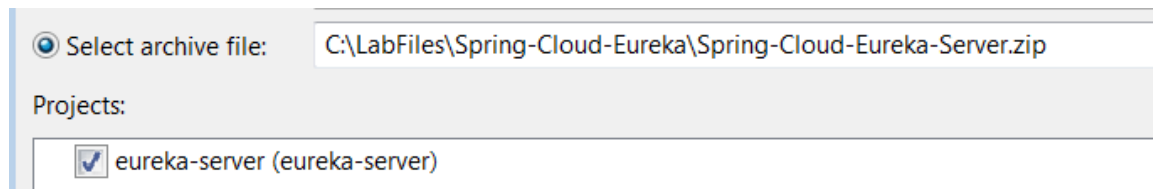
In this lab you will complete a pair of applications; one of them is a service that advertises itself with the Netflix Eureka service. The other is a client that will query the Eureka service to find its server and invoke a RESTful API.

### Part 1 - Run the Eureka Server

The Netflix Eureka Server is available as a ".war" file that you can deploy to a JEE application server (e.g. TomEE). However, it's also quite easy to create a Spring Boot application that runs it. It would be possible to embed a Eureka server with one of your microservices or other infrastructure, but in most cases you'll probably run the Eureka service on its own.

There is a project provided for you that runs the Eureka server. In the part that follows, we'll load it up and examine it, then startup the service.

- \_\_\_ 1. From Eclipse's main menu, click **File** → **Import** and then select **General** → **Existing Projects into Workspace**, and then click **Next**.
- \_\_\_ 2. Click the radio button for **Select archive file**:
- \_\_\_ 3. Click the **Browse** button and navigate to select **C:\LabFiles\Spring-Cloud-Eureka\Spring-Spring-Cloud-Eureka-Server.zip** and then click **Open**.



- \_\_\_ 4. On the **Import** dialog, leave the defaults as-is and click **Finish**.

### Part 2 - Examine the Server Project

There really is very little here. The main thing that runs the embedded Eureka server is the 'EurekaServerApp' class, which you'll recognize as a regular Spring Boot application:

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaServerApp {

    public static void main(String[] args) {
        SpringApplication.run(EurekaServerApp.class, args);
    }
}
```



Note the '@EnableEurekaServer' annotation; it tells Spring Boot to setup the Eureka server.

There are a few items setup in the starter project that enable running the server:

- 'pom.xml' contains the dependency management and dependencies to connect the project to Spring Cloud Netflix and use the Eureka server.

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-netflix</artifactId>
      <version>1.2.3.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka-server</artifactId>
  </dependency>
</dependencies>
```

- There is a file called 'application.yml' in 'src/main/resources' that configures the application:

```
server:
  port: 8761

eureka:
  client:
    registerWithEureka: false
    fetchRegistry: false
  server:
    waitTimeInMsWhenSyncEmpty: 0
```

## Part 3 - Build and Run the Eureka Server

\_\_1. In the **Project Explorer**, right-click on the project node for 'eureka-server' and then select **Run as** → **Maven install**.

At this point, we could run the server inside Eclipse, by running 'EurekaServer.app' as a Java Application. But if we did that, the console output from Eureka might cause distraction during later testing (Eclipse's default behavior is to switch to a console when it has output). So we'll run Eureka from the command line instead.

\_\_2. Open a **Windows Command Prompt**. You can generally find this in the **Start** menu under **All Programs** → **Accessories**.

\_\_3. Enter the following and then press **Return**, to change directory to the Eclipse project where we built Eureka

```
cd \workspace\eureka-server\target
```

\_\_4. Enter the following and then press **Return**.

```
java -Xmx32m -jar eureka-server-0.0.1-SNAPSHOT.jar
```

Note: The '-Xmx32m' option limits the heap size to 32Mb, because we're going to be running several JVMs in this lab. The '-jar...' option runs the jar file that was created by the Spring Boot plugin.

\_\_5. You may be requested to allow access to continue.

\_\_6. You should see the Eureka server start up without any errors.

```
2016-12-13 09:41:26.752 INFO 3012 --- [           main] c.n.e.EurekaDiscoveryCl
ientConfiguration : Updating port to 8761
2016-12-13 09:41:26.755 INFO 3012 --- [           main] com.webage.EurekaServer
App               : Started EurekaServerApp in 16.646 seconds (JVM running for 1
7.201)
```

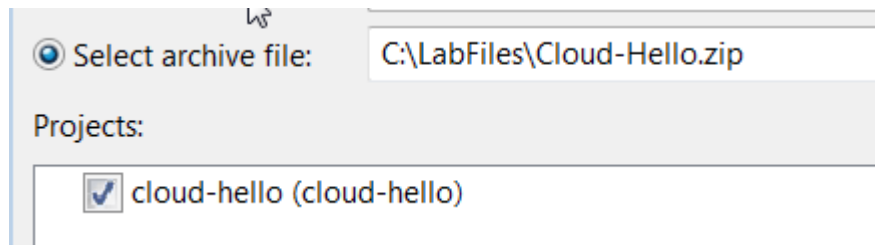
## Part 4 - Run a Eureka-Enabled API

We're going to run a sample client that is already setup to register with Eureka. This client has a couple of features that we'll explore in later labs as well, but for now, we're just looking at its registration behavior.

\_\_1. From Eclipse's main menu, click **File** → **Import** and then select **General** → **Existing Projects into Workspace**, and then click **Next**.

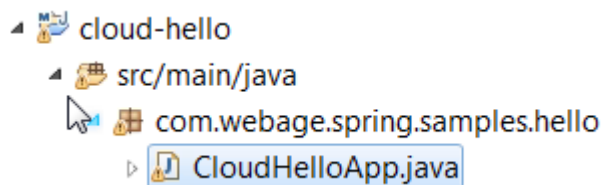
\_\_2. Click the radio button for **Select archive file**:

\_\_3. Click the **Browse** button and navigate to select **C:\LabFiles\Cloud-Hello.zip** and then click **Open**.



\_\_4. On the **Import** dialog, leave the defaults as-is and click **Finish**.

\_\_5. In the **Project Explorer**, locate the class 'CloudHelloApp' in the 'com.webage.spring.samples.hello' package. Double-click on the class to open it.



\_\_6. Note the annotation on the main class:

```
@SpringBootApplication
@EnableEurekaClient
public class CloudHelloApp {

    public static void main(String[] args) {
        SpringApplication.run(CloudHelloApp.class, args);
    }
}
```

The '@EnableEurekaClient' annotation tells Spring Boot to register the application with Eureka.

\_\_7. Close the file 'CloudHelloApp.java'.

\_\_8. Locate the file 'application.yml' in the 'src/main/resources' folder and double-click to open it.

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka

spring:
  application:
    name: cloud-hello
```

```

failAfter:
  enabled=false
  count=100

serverId: 'Cloud-Hello'

logging:
  level:
    root: 'ERROR'
    org.springframework: 'ERROR'

```

There are a few things to observe here:

- The 'eureka/client' section tells Eureka where to find the Eureka server
- The 'spring/application' assigns a name to the client. Clients to this API will look up the API by this name.
- The 'failAfter' section will be used later, when we go to demonstrate the 'circuit-breaker' behavior in Hystrix.
- 'serverId' assigns a server identity that we will override with command-line values when we start multiple servers.
- The 'logging' section configures the root logging category to only log ERROR and higher messages. This will clean up the output significantly.

\_\_9. Close the file 'application.yml'.

\_\_10. In the **Project Explorer**, right-click on the **cloud-hello** project node and then select **Run As → Maven install**. This will build the project, and should complete without errors.

Once again, we're going to run the 'cloud-hello' servers from the command line, so as to leave the Eclipse console clear for the client program that we want to observe.

\_\_11. Open a **Windows Command Prompt** and then type the following to get to the cloud-hello target folder:

```
cd \workspace\cloud-hello\target
```

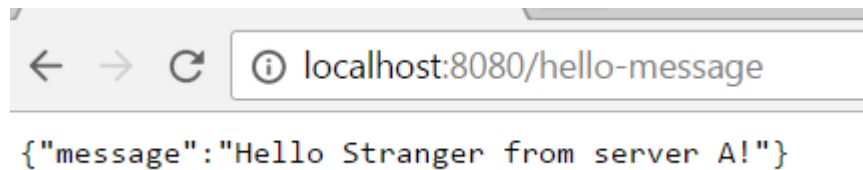
\_\_12. Start a cloud-hello server by typing the following:

```
java -Xmx32m -jar cloud-hello-0.0.1-SNAPSHOT.jar --server.port=8080
--serverId=A
```

(Note the two dashes '--' on the 'server.port' and 'serverId' options!)

\_\_13. You should see the server start up with no errors. To be extra sure, open a web browser and navigate to 'http://localhost:8080/hello-message'

You should see a JSON response from the hello server.



\_\_14. Open a web browser and enter 'http://localhost:8761' into the location bar.

You should see the Eureka Server page, showing the 'CLOUD-HELLO' service with one instance registered.

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
CLOUD-HELLO	n/a (1)	(1)	UP (1) - localhost:cloud-hello:8080

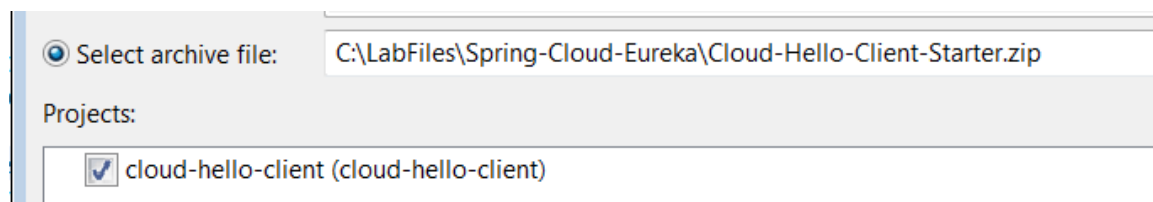
## Part 5 - Complete a Eureka-Enabled Client

So, now we have a Eureka server and one instance of our 'cloud-hello' API registered. Now we'll look at how a client to the API can find an instance to talk to.

\_\_1. From Eclipse's main menu, click **File** → **Import** and then select **General** → **Existing Projects into Workspace**, and then click **Next**.

\_\_2. Click the radio button for **Select archive file**:

\_\_3. Click the **Browse** button and navigate to select **C:\LabFiles\Spring-Cloud-Eureka\Cloud-Hello-Client-Starter.zip** and then click **Open**.

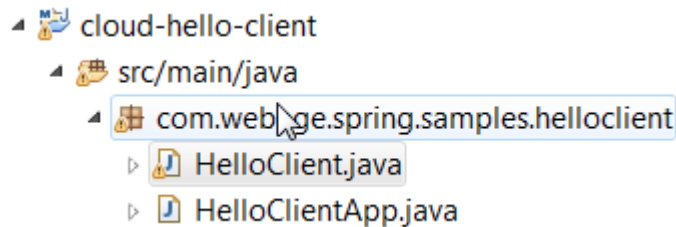


\_\_4. On the **Import** dialog, leave the defaults as-is and click **Finish**.

There are a few things to notice about the project before we start:

- The 'pom.xml' file includes a dependency on the 'spring-cloud-starter-eureka' artifact.
- The 'HelloClientApp' class, which is the Spring Boot startup class, includes the '@EnabledDiscoveryClient' annotation. This tells Spring Boot to setup the Eureka client system.

\_\_5. Locate the file 'HelloClient.java' in the package 'com.webage.spring.samples.helloclient' and double-click the file to open it.



Note the following definition:

```
@Autowired
private DiscoveryClient discoveryClient;
```

This definition requests injection of the 'discovery client' - I.e. the link to Eureka Server.

\_\_6. Insert the following code, after the line that reads '//Insert Lookup Code here...'

```
// Insert Lookup Code here...
List<ServiceInstance> helloServers=
    discoveryClient.getInstances("CLOUD-HELLO");
System.out.println("The following hello-services are available:");
for(ServiceInstance instance: helloServers) {
    System.out.printf("%s at %s\n", instance.getServiceId(),
        instance.getUri().toString());
}
```

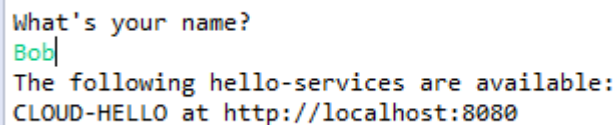
\_\_7. Save the file by pressing **Ctrl-Shift-S**.

\_\_8. Right-click on 'cloud-hello-client' and then select **Run as → Maven install**.

\_\_9. Right-click on the 'HelloClientApp' class and then select **Run As → Java Application**.

\_\_10. Watch the console - the program will ask "What's your name?". Enter a name and then press Return.

\_\_11. You should see output like the following:



```
What's your name?
Bob
The following hello-services are available:
CLOUD-HELLO at http://localhost:8080
```

This tells you that the client was able to discover the cloud-hello API that's running on port 8080. Let's start up another one and see what happens...

\_\_12. Open a **Windows Command Prompt** and then type the following to get to the cloud-hello target folder:

```
cd \workspace\cloud-hello\target
```

\_\_13. Start a cloud-hello server by typing the following:

```
java -Xmx32m -jar cloud-hello-0.0.1-SNAPSHOT.jar --server.port=8081  
--serverId=B
```

(Notice that this time, we set 'server.port' to '8081' and 'serverId' to 'B'. Also remember the two dashes '--' on the 'server.port' and 'serverId' options!)

\_\_14. Right-click on the 'HelloClientApp' class and then select **Run As → Java Application**.

\_\_15. Watch the console - the program will ask "What's your name?". Enter a name and then press Return.

\_\_16. You should see output like the following:

```
What's your name?  
Bob|  
The following hello-services are available:  
CLOUD-HELLO at http://localhost:8081  
CLOUD-HELLO at http://localhost:8080
```

As you can see, we now have two API instances that were discovered.

Note that there is a time delay in registering a service and seeing its availability. So if you see only one service, try running the client again after about a minute. If you still don't see a second instance, contact your instructor.

## Part 6 - Complete the API Call

Now that we've discovered the services, we can go ahead and complete the API call.

\_\_1. In the 'HelloClient.java' file, add the following, just under the line that reads '//Insert code to make the call here...'. For convenience, this code is available in the file 'C:\LabFiles\Spring-Cloud-Eureka\APICall.txt', if you'd prefer to copy and paste it.

```
// Insert code to make the call here...
if(helloServers.size() == 0) {
    System.out.println("Sorry, but I can't find a server to use");
    System.exit(0);
}
// Pick one
int chosen=(int) (Math.random()*helloServers.size());
ServiceInstance chosenInstance=helloServers.get(chosen);
RestTemplate rt=restTemplateBuilder.build();
URI callUri=
    UriComponentsBuilder.fromUri(chosenInstance.getUri())
        .path("/hello-message").queryParams("name", name).build().toUri();
// Do the call.
Map<String, Object> resp=
    (Map<String, Object>) rt.getForObject(callUri, Map.class);
System.out.println("Server says:" + resp.get("message"));
```

\_\_2. Save the file.

\_\_3. Using the same techniques as before, run a Maven install on the 'cloud-hello-client' project.

\_\_4. Right-click on the 'HelloClientApp' class and then select **Run As → Java Application**.

\_\_5. Watch the console - the program will ask "What's your name?". Enter a name and then press Return.

\_\_6. You should now see output something like this:

```
What's your name?
Bob
The following hello-services are available:
CLOUD-HELLO at http://localhost:8081
CLOUD-HELLO at http://localhost:8080
Server says:Hello Bob from server B!
```

\_\_7. If you run the program a few more times, you should see that we're selecting a server randomly between server A and server B.

\_\_8. Leave the 'cloud-hello' servers and the Eureka servers running - we'll use them in the next lab.

\_\_9. Close the editors.



## **Part 7 - Review**

In this lab, we used Spring Cloud's utilities to run a Eureka server and then we created a client that registered itself with the Eureka service registrar. Then we created a client that used Spring Cloud's Eureka client support to find all the instances of the server it needed, and make a RESTful API call on a randomly-chosen service.

## Lab 12 - Use Netflix Ribbon for Client-Side Load Balancing

In this lab we will start up a suite of server application instances and complete a client that load-balances requests between them.

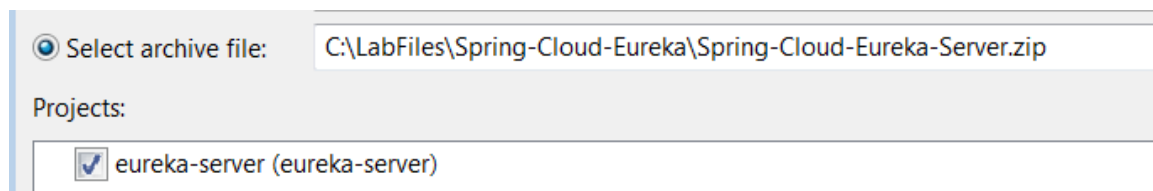
### Part 1 - Run the Eureka Server

Note: If you recently completed the Spring Cloud Eureka lab, you may have the Eureka server already running. If so, you can leave it running and skip this Part.

The Netflix Eureka Server is available as a ".war" file that you can deploy to a JEE application server (e.g. TomEE). However, it's also quite easy to create a Spring Boot application that runs it. It would be possible to embed a Eureka server with one of your microservices or other infrastructure, but in most cases you'll probably run the Eureka service on its own.

There is a project provided for you that runs the Eureka server. In the part that follows, we'll load it up and examine it, then startup the service.

- \_\_1. From Eclipse's main menu, click **File** → **Import** and then select **General** → **Existing Projects into Workspace**, and then click **Next**.
- \_\_2. Click the radio button for **Select archive file**:
- \_\_3. Click the **Browse** button and navigate to select **C:\LabFiles\Spring-Cloud-Eureka\Spring-Cloud-Eureka-Server.zip** and then click **Open**.



- \_\_4. On the **Import** dialog, leave the defaults as-is and click **Finish**.
- \_\_5. In the **Project Explorer**, right-click on the project node for 'eureka-server' and then select **Run as** → **Maven install**.

At this point, we could run the server inside Eclipse, by running 'EurekaServer.app' as a Java Application. But if we did that, the console output from Eureka might cause distraction during later testing (Eclipse's default behavior is to switch to a console when it has output). So we'll run Eureka from the command line instead.

- \_\_6. Open a **Windows Command Prompt**. You can generally find this in the **Start** menu under **All Programs** → **Accessories**.

\_\_7. Enter the following and then press **Return**, to change directory to the Eclipse project where we built Eureka

```
cd \workspace\eureka-server\target
```

\_\_8. Enter the following and then press **Return**.

```
java -Xmx32m -jar eureka-server-0.0.1-SNAPSHOT.jar
```

Note: The '-Xmx32m' option limits the heap size to 32Mb, because we're going to be running several JVMs in this lab. The '-jar...' option runs the jar file that was created by the Spring Boot plugin.

\_\_9. You should see the Eureka server start up without any errors.

## Part 2 - Run a Pair of Cloud-Hello Instances

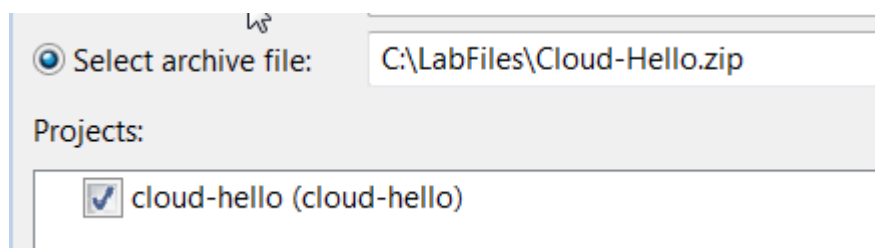
**Note:** If you recently completed the Spring Cloud Eureka lab, you may have two instances of Cloud-Hello already running. If so, you can leave them running and skip this Part.

We're going to run a sample client that is already setup to register with Eureka. This client has a couple of features that we'll explore in later labs as well, but for now, we're just looking at its registration behavior.

\_\_1. From Eclipse's main menu, click **File** → **Import** and then select **General** → **Existing Projects into Workspace**, and then click **Next**.

\_\_2. Click the radio button for **Select archive file**:

\_\_3. Click the **Browse** button and navigate to select **C:\LabFiles\Cloud-Hello.zip** and then click **Open**.



\_\_4. On the **Import** dialog, leave the defaults as-is and click **Finish**.

\_\_5. In the **Project Explorer**, right-click on the **cloud-hello** project node and then select **Run As** → **Maven install**. This will build the project, and should complete without errors.

Once again, we're going to run the 'cloud-hello' servers from the command line, so as to leave the Eclipse console clear for the client program that we want to observe.

\_\_6. Open a **Windows Command Prompt** and then type the following to get to the cloud-hello target folder:

```
cd \workspace\cloud-hello\target
```

\_\_7. Start a cloud-hello server by typing the following:

```
java -Xmx32m -jar cloud-hello-0.0.1-SNAPSHOT.jar --server.port=8080  
--serverId=A
```

(Note the two dashes '--' on the 'server.port' and 'serverId' options!)

\_\_8. You should see the server start up with no errors. To be extra sure, open a web browser and navigate to 'http://localhost:8080/hello-message'. You should see a JSON response from the hello server.

\_\_9. Open another **Windows Command Prompt** and then type the following to get to the cloud-hello target folder:

```
cd \workspace\cloud-hello\target
```

\_\_10. Start a cloud-hello server by typing the following:

```
java -Xmx32m -jar cloud-hello-0.0.1-SNAPSHOT.jar --server.port=8081  
--serverId=B
```

(Notice that this time, we set 'server.port' to '8081' and 'serverId' to 'B'. Also remember the two dashes '--' on the 'server.port' and 'serverId' options!)

\_\_11. Open a web browser and enter 'http://localhost:8761' into the location bar. You should see the Eureka Server page, showing the 'CLOUD-HELLO' service with two instances registered.

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
CLOUD-HELLO	n/a (2)	(2)	UP (2) - localhost:cloud-hello:8081 , localhost:cloud-hello:8080

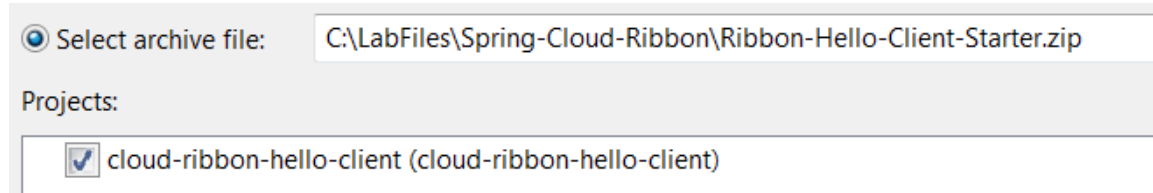
## Part 3 - Complete a Ribbon-Enabled Client

So, now we have a Eureka server and two instance of our 'cloud-hello' API registered. Now we'll look at how to use the Spring Cloud Ribbon technology to perform client-side load balancing between them

\_\_\_1. From Eclipse's main menu, click **File** → **Import** and then select **General** → **Existing Projects into Workspace**, and then click **Next**.

\_\_\_2. Click the radio button for **Select archive file**:

\_\_\_3. Click the **Browse** button and navigate to select **C:\LabFiles\Spring-Cloud-Ribbon\Ribbon-Hello-Client-Starter.zip** and then click **Open**.

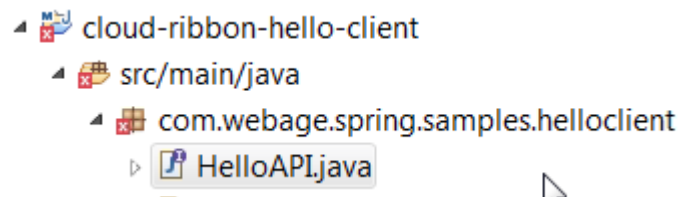


\_\_\_4. On the **Import** dialog, leave the defaults as-is and click **Finish**.

There are a few things to notice about the project before we start:

- The 'pom.xml' file includes a dependency on the 'spring-cloud-starter-eureka', 'spring-cloud-starter-ribbon' and 'spring-cloud-starter-feign' artifacts.
- The 'RibbonHelloClientApp' class, which is the Spring Boot startup class, includes the '[@EnableDiscoveryClient](#)' annotation. This tells Spring Boot to setup the Eureka client system.
- The 'RibbonHelloClientApp' class, which is the Spring Boot startup class, includes the '[@EnableFeignClients](#)' annotation. This tells Spring Boot to setup the proxy-based 'Feign' client system.

\_\_\_5. Locate the file 'HelloAPI' in the package 'com.webage.spring.samples.helloclient' and double-click the file to open it.



\_\_6. For reference, it's reproduced here:

```
@FeignClient(value="cloud-hello")
public interface HelloAPI {
    @RequestMapping(value="/hello-message", method=RequestMethod.GET)
    public Map<String, Object>
        getGreeting(@RequestParam("name") String name);
}
```

The interface represents a way that we can call the cloud-hello API that we started up earlier in the lab. Notice that the interface is annotated in much the same way that an implementation class would be annotated using Spring MVC annotations (the one difference is that the Feign client proxy generator does not recognize the newer '@GetMapping' annotation, so we need to use the older '@RequestMapping(method=RequestMethod.GET...)' style of annotation).

The Feign proxy subsystem will collaborate with Ribbon to generate a class that implements this interface by calling the discovered services, and it will make that proxy available for dependency injection.

\_\_7. Locate the file 'RibbonHelloClient.java' in the package 'com.webage.spring.samples.helloclient' and double-click the file to open it.

\_\_8. Add the following code just after the line that reads '// Insert autowired HelloAPI here...'

```
// Insert autowired HelloAPI here...
@Autowired
HelloAPI helloAPI;
```

This definition requests injection of the generated proxy to the Cloud-Hello API.

\_\_9. Insert the following code, after the line that reads '// Insert code to make the call here...'

```
// Insert code to make the call here...
Map<String, Object> resp = helloAPI.getGreeting(name);
String greeting=(String) resp.get("message");
System.out.println("Server says:" + resp.get("message"));
```

As you can see, there's quite a bit less code required, as compared to the RestTemplate approach. We simply call the proxy's 'getGreeting' method, receiving a Map object that is a good model for the JSON object that we'll receive from the server.

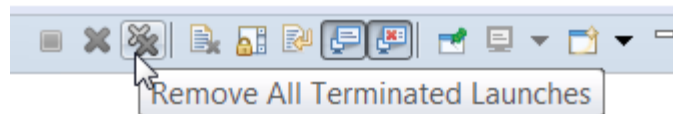
The rest of the code counts how many of each unique greeting we get back. Recall that we started up the cloud-hello API instances with a distinct 'serverId' value for each instance. The response messages will indicate which server actually handled the messages.

- \_\_10. Save the file by pressing **Ctrl-Shift-S**.
- \_\_11. Right-click on 'cloud-ribbon-hello-client' and then select **Run as → Maven install**.
- \_\_12. Right-click on the 'RibbonHelloClientApp' class and then select **Run As → Java Application**.
- \_\_13. Watch the console - the program will ask "What's your name?". Enter a name and then press Return.
- \_\_14. You should see output like the following:

```
Server says:Hello Bob from server B!  
Server says:Hello Bob from server A!  
Server says:Hello Bob from server B!  
Server says:Hello Bob from server A!  
Hello Bob from server B! occurred 250 times  
Hello Bob from server A! occurred 250 times  
Press any key to exit
```

This tells you that server 'A' and server 'B' each processed 250 of the 500 requests.

- \_\_15. Press any click inside the Console to terminate the process.
- \_\_16. Click the icon below as many times until the Console is clean.



- \_\_17. Shut down the two 'cloud-hello' servers by closing each of their command prompt windows but leave the eureka running.

```
Command Prompt - java -Xmx32m -jar eureka-server-0.0.1-SNAPSHOT.jar  
2016-12-13 13:34:26.369 INFO 3012 ---[a-EvictionTimer] c.n.e.re  
InstanceRegistru : Running the evict task with compensationTime
```

- \_\_18. Close any open browser.
- \_\_19. Close all open editors.

## Part 4 - Review

In this lab, we used Spring Cloud's support for Netflix Ribbon to perform client-side load balancing between a pair of servers.

## Lab 13 - Use Netflix Hystrix for the Circuit Breaker Pattern

In this lab we will start up a suite of server application instances and complete a client that load-balances requests between them, but also incorporates the "Circuit-Breaker" pattern with fallback behavior when the servers fail.

### Part 1 - Run the Eureka Server

Note: If you recently completed the Spring Cloud Eureka lab, you may have the Eureka server already running. If so, you can leave it running and skip this Part.

The Netflix Eureka Server is available as a ".war" file that you can deploy to a JEE application server (e.g. TomEE). However, it's also quite easy to create a Spring Boot application that runs it. It would be possible to embed a Eureka server with one of your microservices or other infrastructure, but in most cases you'll probably run the Eureka service on its own.

There is a project provided for you that runs the Eureka server. In the part that follows, we'll load it up and examine it, then startup the service.

- \_\_1. From Eclipse's main menu, click **File** → **Import** and then select **General** → **Existing Projects into Workspace**, and then click **Next**.
- \_\_2. Click the radio button for **Select archive file**:
- \_\_3. Click the **Browse** button and navigate to select **C:\LabFiles\Spring-Cloud-Eureka\Spring-Cloud-Eureka-Server.zip** and then click **Open**.
- \_\_4. On the **Import** dialog, leave the defaults as-is and click **Finish**.
- \_\_5. In the **Project Explorer**, right-click on the project node for 'eureka-server' and then select **Run as** → **Maven install**.

At this point, we could run the server inside Eclipse, by running 'EurekaServer.app' as a Java Application. But if we did that, the console output from Eureka might cause distraction during later testing (Eclipse's default behavior is to switch to a console when it has output). So we'll run Eureka from the command line instead.

- \_\_6. Open a **Windows Command Prompt**. You can generally find this in the **Start** menu under **All Programs** → **Accessories**.
- \_\_7. Enter the following and then press **Return**, to change directory to the Eclipse project where we built Eureka

```
cd \workspace\eureka-server\target
```



\_\_ 8. Enter the following and then press **Return**.

```
java -Xmx32m -jar eureka-server-0.0.1-SNAPSHOT.jar
```

Note: The '-Xmx32m' option limits the heap size to 32Mb, because we're going to be running several JVMs in this lab. The '-jar...' option runs the jar file that was created by the Spring Boot plugin.

\_\_ 9. You should see the Eureka server start up without any errors.

## Part 2 - Run a Pair of Cloud-Hello Instances

**Note:** If you recently completed the Spring Cloud Eureka lab, you may have two instances of Cloud-Hello already running. We need to start them up with different options, so if you have any 'Cloud-Hello' instances running, shut them down by closing their command prompt windows.

We're going to run a sample client that is already setup to register with Eureka. This client has a feature whereby we can tell it to fail after a certain number of calls in a 30-second period. We'll use this feature to simulate a failure and demonstrate the "circuit breaker" pattern.

Once again, we're going to run the 'cloud-hello' servers from the command line, so as to leave the Eclipse console clear for the client program that we want to observe.

\_\_ 1. Make sure there are not cloud-hello instances running.

\_\_ 2. Open a **Windows Command Prompt** and then type the following to get to the cloud-hello target folder:

```
cd \workspace\cloud-hello\target
```

\_\_ 3. Start a cloud-hello server by typing the following:

```
java -Xmx32m -jar cloud-hello-0.0.1-SNAPSHOT.jar --server.port=8080  
--serverId=A --failAfter.enabled=true
```

The 'failAfter.enabled=true' option turns on the simulated failure behavior. (Note the two dashes '--' on the 'server.port', 'serverId' and 'failAfter.enabled' options!)

\_\_ 4. You should see the server start up with no errors. To be extra sure, open a web browser and navigate to 'http://localhost:8080/hello-message'. You should see a JSON response from the hello server.

\_\_5. Open another **Windows Command Prompt** and then type the following to get to the cloud-hello target folder:

```
cd \workspace\cloud-hello\target
```

\_\_6. Start a cloud-hello server by typing the following:

```
java -Xmx32m -jar cloud-hello-0.0.1-SNAPSHOT.jar --server.port=8081  
--serverId=B --failAfter.enabled=true
```

(Notice that this time, we set 'server.port' to '8081' and 'serverId' to 'B'. Also remember the two dashes '--' on the 'server.port', 'serverId' and 'failAfter.enabled' options!))

\_\_7. Open a web browser and enter 'http://localhost:8761' into the location bar. You should see the Eureka Server page, showing the 'CLOUD-HELLO' service with two instances registered.

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
CLOUD-HELLO	n/a (2)	(2)	UP (2) - localhost:cloud-hello:8081 , localhost:cloud-hello:8080

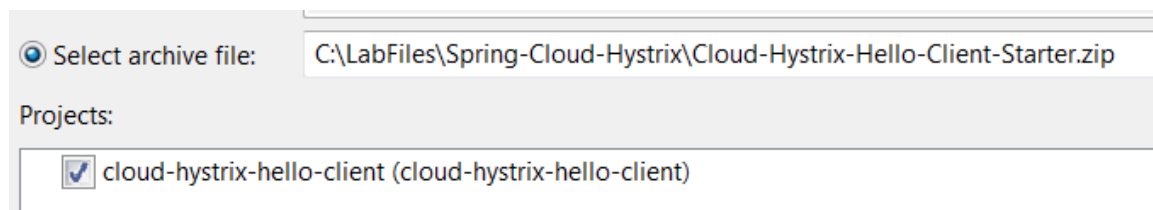
### Part 3 - Complete a Hystrix-Enabled Client

So, now we have a Eureka server and two instance of our 'cloud-hello' API registered. Now we'll look at how to use the Spring Cloud Ribbon technology to perform client-side load balancing between them

\_\_1. From Eclipse's main menu, click **File** → **Import** and then select **General** → **Existing Projects into Workspace**, and then click **Next**.

\_\_2. Click the radio button for **Select archive file**:

\_\_3. Click the **Browse** button and navigate to select **C:\LabFiles\Spring-Cloud-Hystrix\Cloud-Hystrix-Hello-Client-Starter.zip** and then click **Open**.

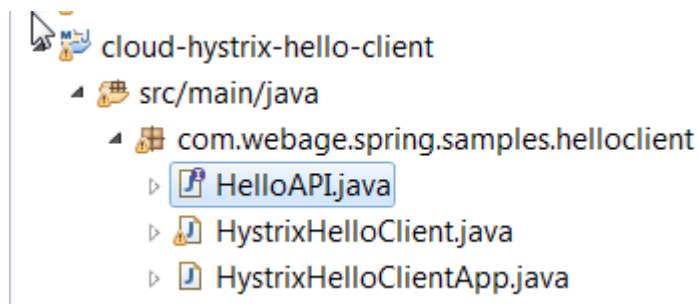


\_\_4. On the **Import** dialog, leave the defaults as-is and click **Finish**.

\_\_5. There are a few things to notice about the project before we start:

- The 'pom.xml' file includes a dependency on the 'spring-cloud-starter-eureka', 'spring-cloud-starter-ribbon' and 'spring-cloud-starter-feign' artifacts.
- The 'HystrixHelloClientApp' class, which is the Spring Boot startup class, includes the '@EnableDiscoveryClient' annotation. This tells Spring Boot to setup the Eureka client system.
- The 'HystrixHelloClientApp' class, which is the Spring Boot startup class, includes the '@EnableFeignClients' annotation. This tells Spring Boot to setup the proxy-based 'Feign' client system.
- 'HystrixHelloClient' also has the '@EnableCircuitBreaker' annotation, which enables Hystrix.

\_\_6. Locate the file 'HelloAPI' in the package 'com.webage.spring.samples.helloclient' and double-click the file to open it.



For reference, it's reproduced here:

```
@FeignClient(value="cloud-hello")
public interface HelloAPI {
    @RequestMapping(value="/hello-message", method=RequestMethod.GET)
    public Map<String, Object>
        getGreeting(@RequestParam("name") String name);
}
```

The interface represents a way that we can call the cloud-hello API that we started up earlier in the lab. Notice that the interface is annotated in much the same way that an implementation class would be annotated using Spring MVC annotations (the one difference is that the Feign client proxy generator does not recognize the newer '@GetMapping' annotation, so we need to use the older '@RequestMapping(method=RequestMethod.GET...)' style of annotation).

The Feign proxy subsystem will collaborate with Ribbon to generate a class that implements this interface by calling the discovered services, and it will make that proxy available for dependency injection.

\_\_7. Modify the '@FeignClient' annotation to read as follows:

```
@FeignClient(value="cloud-hello", fallback=HelloAPIFallback.class)
```

This edit establishes that if the remote API fails for any reason, the methods in 'HelloAPIFallback' can be used instead. You will have an error showing for now, because we haven't yet created the 'HelloAPIFallback' class.

\_\_8. Organize imports.

\_\_9. Save the class. You will see an error.

\_\_10. Right-click on the 'com.webage.spring.samples.helloclient' and then select **New** → **Class**.

\_\_11. Enter 'HelloAPIFallback' as the class name and click **Finish**.

\_\_12. Edit the main body of the new class to read as follows.

```
@Component
public class HelloAPIFallback implements HelloAPI {

    @Override
    public Map<String, Object> getGreeting(String name) {
        Map<String, Object> resp=new HashMap<String, Object>();
        resp.put("message", "Fallback");
        return resp;
    }
}
```

\_\_13. Organize imports.

\_\_14. Save and close the file. The error in HelloAPI will gone.

\_\_15. Right-click on 'cloud-hystrix-hello-client' and then select **Run as** → **Maven install**.

\_\_16. Right-click on the 'HystrixHelloClientApp' class and then select **Run As** → **Java Application**.

\_\_17. Watch the console - the program will ask "What's your name?". Enter a name and then press Return.

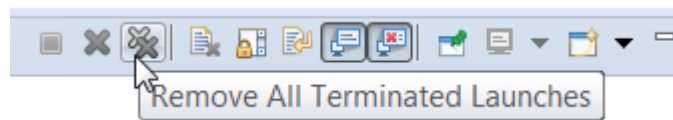
\_\_18. You should see output like the following:

```
Server says:Fallback
Server says:Fallback
Server says:Fallback
Server says:Fallback
Hello Bob from server B! occurred 100 times
Fallback occurred 300 times
Hello Bob from server A! occurred 100 times
```

This tells you that server 'A' and server 'B' each processed 100 requests and then "failed". The fallback class was used 300 times. Your result may vary a little bit.

\_\_19. Press return to exit the console.

\_\_20. Click the icon below as many times until the Console is clean.



\_\_21. If you'd like to run the program again, wait about 30 seconds for the simulated failures to reset.

\_\_22. Close all open files.

\_\_23. Close all browsers.

\_\_24. Close all command prompt windows.

## Part 4 - Review

In this lab, we used Spring Cloud's support for Netflix Hystrix to provide a circuit-breaker with fallback for the remote API.

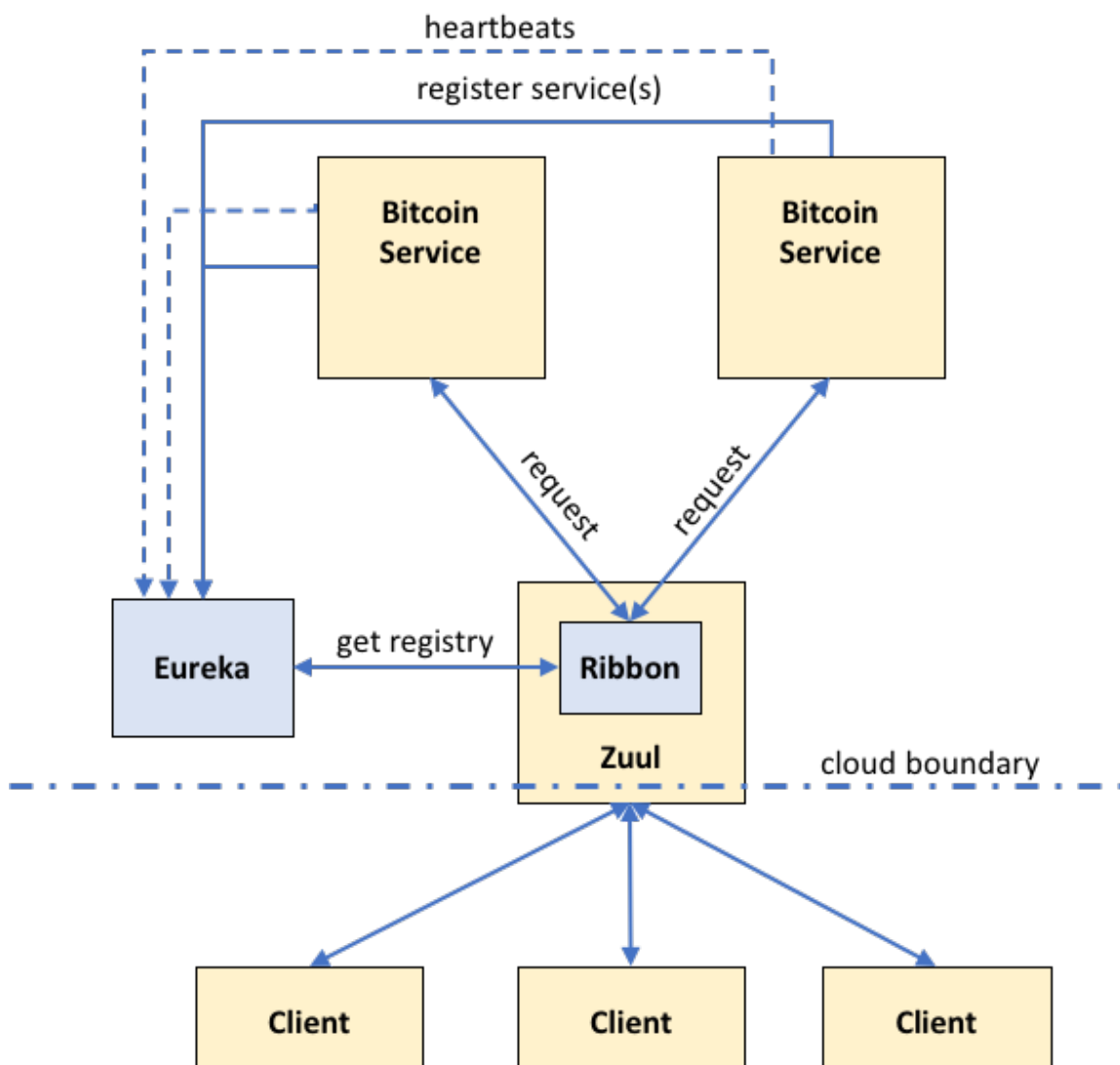
## Lab 14 - EdgeComponents with Zuul

In this lab we will use Zuul as a gateway to access a service that gives the current price of Bitcoin in dollars. (Since the value of Bitcoin varies is volatile, our service will return volatile values as well.) We will start and stop multiple instances of the Bitcoin service,so our Zuul gateway will use Eureka and Ribbon for service discovery and load balancing.

Since Zuul is an edge service, all client requests go through it. Therefore, we will take advantage of Zuul filters to log incoming requests and log elapsed time for each request.

We will also start and stop the Bitcoin services to demonstrate the ability of Zuul/Ribbon/Eureka to respond to changes in server health and availability.

This is a high-level diagram of what we will be building in this lab.



## Part 1 - Setup

In this part we will perform the basic setup necessary to complete this lab.

- \_\_1. Make sure that no Spring Boot applications are running.
- \_\_2. Examine the C:\LabFiles\Spring-Cloud-Zuul directory, you should see three directories: gateway, eureka-service and bitcoin-price-service. These directory contain the partially completed Spring Boot applications that we will use in this lab.

NOTE: for simplicity in this lab we will use the conventions

Shortcut	Path
<bitcoin>	C:\LabFiles\Spring-Cloud-Zuul\bitcoin-price-service
<eureka>	C:\LabFiles\Spring-Cloud-Zuul\eureka-service
<gateway>	C:\LabFiles\Spring-Cloud-Zuul\gateway

## Part 2 - The Bitcoin Service

The Bitcoin service is a RESTful Spring Boot service that creates returns the current price of Bitcoin in dollars. The service chooses a random price between \$1 and \$20000 and returns it as a string (including the dollar sign.)

- \_\_1. Open <bitcoin>\src\main\java\bitcoin\BitcoinPriceService.java in a text editor or IDE
- \_\_2. Add the following class level annotations to **BitcoinPriceService**.

```
@RestController
@SpringBootApplication
```

- \_\_3. Add a main method that calls `SpringApplication.run()`

```
public static void main(String[] args) {
    SpringApplication.run(BitcoinPriceService.class, args);
}
```

4. Add a method that returns a random String between "\$1" and "\$20000".

```
public String getPrice() {
    int price = ThreadLocalRandom.current().nextInt(1, 20000+1);
    String priceString = "$"+price;
    log.info(priceString);
    return priceString;
}
```

5. Map the method to `/price`

```
@RequestMapping("/price")
```

6. Save the file.

**7. Open <bitcoin>\src\main\resources\bootstrap.properties**

\_\_8. Add the following lines to configure the port and service name

```
server.port=9000
spring.application.name=bitcoin-service
```

9. Save the file.

\_\_10. Open a DOS shell to <bitcoin>

```
cd C:\LabFiles\Spring-Cloud-Zuul\bitcoin-price-service
```

11. Execute this command to start the service

```
mvn spring-boot:run
```

\_\_12. You will probably see several Warnings related to the Eureka service. This is because the POM file declares a dependency on **spring-cloud-starter-eureka** and we have not yet configured the Eureka service. The service will still run even without Eureka.

```
port.decorator.RetryableEurekaHttpClient.execute(Retrya
1.7.0.jar:1.7.0]
port.decorator.EurekaHttpClientDecorator.getApplication
-client-1.7.0.jar:1.7.0]
port.decorator.EurekaHttpClientDecorator$6.execute(Eure
-1.7.0.jar:1.7.0]
port.decorator.SessionedEurekaHttpClient.execute(Sessio
7.0.jar:1.7.0]
port.decorator.EurekaHttpClientDecorator.getApplication
```



\_\_13. Open Postman.

\_\_14. Select **Get**.

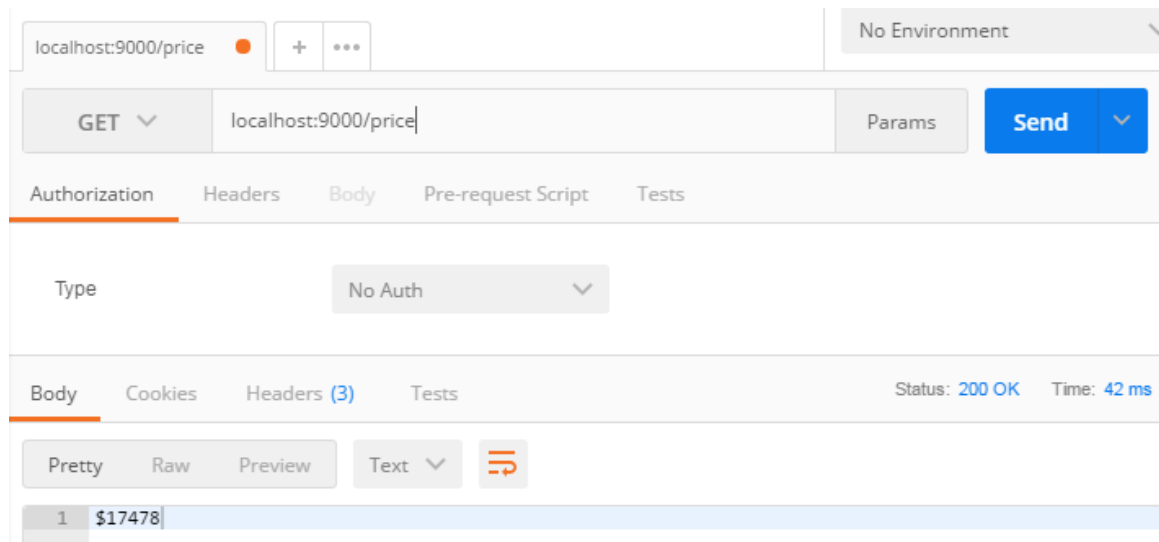
\_\_15. Enter a get request to:

localhost:9000/price

\_\_16. Click **Send**.

You should see a value returned to Postman and the same value in the DOS shell.

```
at java.lang.Thread.run(Thread.java:748)
2018-02-22 11:15:55.218 INFO 1868 --- [nio-9000]
vice : $17478
```



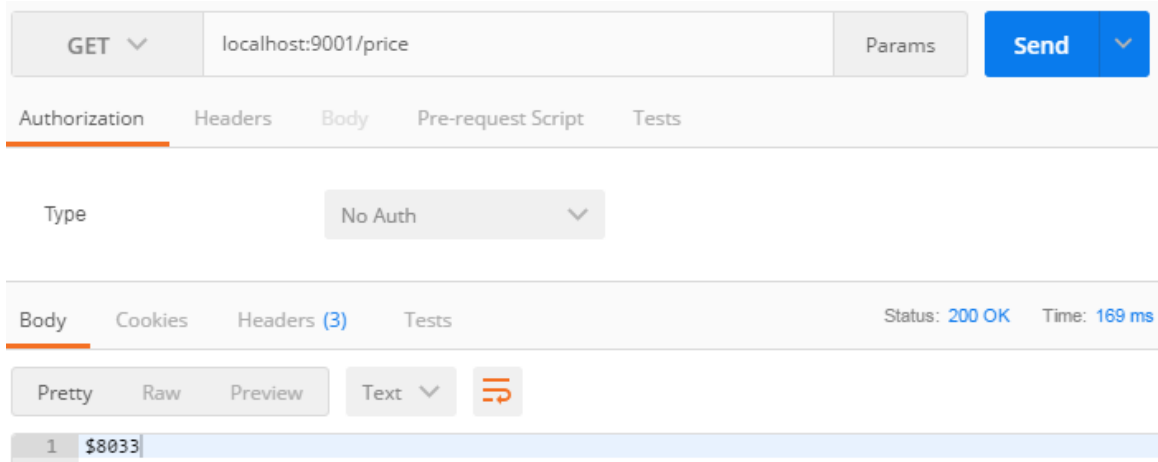
Note, You may need to click on the mvn window and hit enter if the button is shown as 'Sending'.

\_\_17. Click again **Send** and the value will be different each time.

\_\_18. Open another DOS shell and start another instance of the bitcoin service on a different port with the following command:

```
cd C:\LabFiles\Spring-Cloud-Zuul\bitcoin-price-service
mvn spring-boot:run -Drun.arguments="--server.port=9001"
```

\_\_19. Connect to this instance with Postman and you should see a random price for bitcoin.



### Part 3 - The Eureka Service

In this part you will start the Eureka service. The Eureka service is simple so we will not make any changes to it.

- \_\_1. Examine `<eureka>/src/main/java/hello/EurekaServiceApplication.java` in text editor. This is the basic Eureka service that we have seen in previous labs.
- \_\_2. Open `<eureka>/src/main/resources/bootstrao.properties`. It may seem strange that the Eureka features are turned off. They are turned off so that the service does not register with itself recursively.
- \_\_3. Open a DOS shell to `<eureka>`
- \_\_4. Start Eureka with the following command:

```
mvn spring-boot:run
```

- \_\_5. You should see log messages in the two bitcoin service shells indicating that the bitcoin service instances have registered with Eureka.



## Part 4 - Configure Zuul to use Eureka and Ribbon

In this part we will configure the Zuul and use it to load balance request to the two bitcoin service instances.

- \_\_1. Open `<gateway>/src/main/java/hello/GatewayApplication.java`
- \_\_2. The following class level annotations enable Zuul and Eureka integration.

```
@EnableZuulProxy
@EnableDiscoveryClient
```

- \_\_3. Open `<gateway>/src/main/resources/application.properties`
- \_\_4. The following enable Ribbon load-balancing and set the port.

```
ribbon.eureka.enabled=true
server.port=8080
```

- \_\_5. Open `<gateway>/pom.xml` notice the dependencies on zuul and eureka
- \_\_6. Open `<gateway>/src/main/java/hello/filters/pre/StartTimerFilter.java`
- \_\_7. Comment out extends ZuulFilter as shown in bold.

```
public class StartTimerFilter /*extends ZuulFilter*/ {
```

- \_\_8. Save the file.
- \_\_9. Open `<gateway>/src/main/java/hello/filters/post/ElapsedTimeLoggerFilter.java`
- \_\_10. Comment out extends ZuulFilter as shown in bold.

```
public class ElapsedTimeLoggerFilter /*extends ZuulFilter*/ {
```

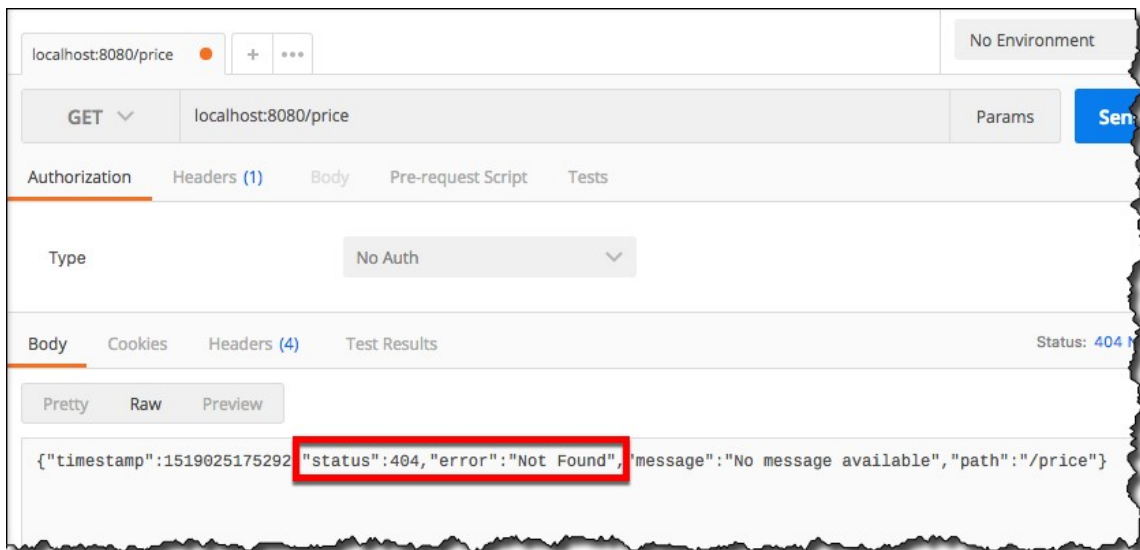
- \_\_11. Save the file.
- \_\_12. Open a DOS shell to `<gateway>`
- \_\_13. Start the Zuul server with the following command:

```
mvn spring-boot:run
```

You should now be able to access bitcoin prices by going through the zuul server. Remember that the zuul server is listening to port 8080

\_\_14. Try accessing bitcoin prices in Postman at **http://localhost:8080/price**

Unfortunately this request fails. The reason is that the Ribbon lay requires that the URL use the service name registered with Eureka.



\_\_15. Open **<bitcoin>/src/main/resources/bootstrap.properties**

\_\_16. Find the value of **spring.application.name** and write it in the blank below to complete the URL

http://localhost:8080/\_\_\_\_\_/price

\_\_17. Use this URL to connect to zuul from Postman. You should get a response.

\_\_18. Make multiple requests from Postman to the same URL. Watch the bitcoin service shells and you should see the requests being load balanced between them.

## Part 5 - Configure Zuul Filters

In this part we will configure three Zuul filters. The first filter will be a pre-filter that logs each request's URI. The second two filters are a pair – one pre-filer and one post-filter. The pre-filter gets the current time in milliseconds and stores it in the context. The post-filter gets the current time and subtracts the uses the time passed in the context to determine and log the elapsed time for this request.

\_\_1. Open **<gateway>/src/main/java/hello/filters/pre/RequestLogger.java**

Notice that this is a "pre" filter, it is #1 in order and it is enabled.

\_\_2. Open <gateway>/src/main/java/hello/GatewayApplication.java

\_\_3. Add the following method to enable the RequestLoggerFilter.

```
@Bean
public RequestLoggerFilter requestLoggerFilter() {
    return new RequestLoggerFilter();
}
```

\_\_4. Save the file.

\_\_5. Shutdown the gateway server with 'Ctrl-C' in the shell.

\_\_6. Restart the gateway server

\_\_7. Use Postman to make a couple requests. You should see the request info logged in the gateway shell:



```
gateway — java -classpath /usr/local/Cellar/maven/3.3.9/libexec/boot/plexus-classworlds-2.5.2.jar -Dclassworlds.conf=/usr/local/Cellar/maven/3.3.9/libexec/bin/m2.conf -D...
2018-02-19 01:53:19.468 INFO 21966 --- [nio-8080-exec-3] hello.filters.pre.RequestLoggerFilter
GET request to http://localhost:8080/bitcoin-service/price
2018-02-19 01:53:21.883 INFO 21966 --- [nio-8080-exec-5] hello.filters.pre.RequestLoggerFilter
GET request to http://localhost:8080/bitcoin-service/price
2018-02-19 01:53:22.223 INFO 21966 --- [nio-8080-exec-6] hello.filters.pre.RequestLoggerFilter
GET request to http://localhost:8080/bitcoin-service/price
2018-02-19 01:53:22.451 INFO 21966 --- [nio-8080-exec-7] hello.filters.pre.RequestLoggerFilter
GET request to http://localhost:8080/bitcoin-service/price
```

\_\_8. Add the following <gateway>/src/main/java/hello/GatewayApplication.java to enable the StartTimerFilter and the ElapsedTimeLoggerFilter:

```
@Bean
public StartTimerFilter startTimerFilter() {
    return new StartTimerFilter();
}

@Bean
public ElapsedTimeLoggerFilter elapsedTimeLoggerFilter() {
    return new ElapsedTimeLoggerFilter();
}
```

\_\_9. Save the file.

\_\_10. Open <gateway>/src/main/java/hello/filters/pre/StartTimerFilter.java

\_\_11. Add the following code to implement the filter:

```
@Override
public String filterType() {
    return "pre";
}

@Override
public int filterOrder() {
    return 2;
}

@Override
public boolean shouldFilter() {
    return true;
}

@Override
public Object run() {
    RequestContext ctx = RequestContext.getCurrentContext();
    // set the current time in ms in the context
    ctx.set("TIME", new Date().getTime());
    return null;
}
```

\_\_12. Save the file.

\_\_13. Open <gateway>/src/main/java/hello/filters/post/ElapsedTimeLoggerFilter.java

\_\_14. Add the following code to implement the filter:

```
@Override
public String filterType() {
    return "post";
}

@Override
public int filterOrder() {
    return 1;
}

@Override
public boolean shouldFilter() {
    return true;
}
```

```

@Override
public Object run() {
    RequestContext ctx = RequestContext.getCurrentContext();

    // get the startTime stored in the context
    long startTime = (long)ctx.get("TIME");
    long currentTime = new Date().getTime();

    // calculate and log the elapsed time
    log.info("elapsed time: " + (currentTime-startTime) + "ms");
    return null;
}

```

- \_\_15. Save the file.
- \_\_16. Shutdown the gateway server with 'Ctrl-C' in the shell
- \_\_17. Open <gateway>/src/main/java/hello/filters/pre/StartTimerFilter.java
- \_\_18. Un-Comment out extends ZuulFilter as shown in bold.

```

public class StartTimerFilter extends ZuulFilter {

```

- \_\_19. Save the file.
- \_\_20. Open <gateway>/src/main/java/hello/filters/post/ElapsedTimeLoggerFilter.java
- \_\_21. Un-Comment out extends ZuulFilter as shown in bold.

```

public class ElapsedTimeLoggerFilter extends ZuulFilter {

```

- \_\_22. Save the file.
- \_\_23. Make sure you saved all files.
- \_\_24. Shutdown the gateway server with 'Ctrl-C' in the shell
- \_\_25. Restart the gateway server
- \_\_26. Use Postman to make a couple requests. You should see the request and elapsed time logged in the gateway shell:

```

gateway — java -classpath /usr/local/Cellar/maven/3.3.9/libexec/boot/plexus-classworlds-2.5.2.jar -Dclassworlds.conf=/usr/local/Cellar/maven/3.3.9/libexec/bin/m2.conf -D...
2018-02-19 02:08:02.618 INFO 22091 --- [nio-8080-exec-2] h.filters.post.ElapsedTimeLoggerFilter :
elapsed time: 15ms
2018-02-19 02:08:05.081 INFO 22091 --- [nio-8080-exec-3] hello.filters.pre.RequestLoggerFilter
GET request to http://localhost:8080/bitcoin-service/price
2018-02-19 02:08:05.092 INFO 22091 --- [nio-8080-exec-3] h.filters.post.ElapsedTimeLoggerFilter
elapsed time: 9ms
2018-02-19 02:08:05.095 INFO 22091 --- [nio-8080-exec-3] h.filters.pre.RequestLoggerFilter

```

## Part 6 - Server Availability

In this part we will shutdown and restart servers to verify that Eureka, Ribbon and Zuul are working together to handle server failures.

- \_\_\_ 1. Shut down one of the Bitcoin servers by entering 'Ctrl-c' in its shell.
- \_\_\_ 2. Send some requests through Postman. They should all be routed to the Bitcoin server that is still running. You may see the first request as error, if so then try another request.
- \_\_\_ 3. Shut down the other Bitcoin server.
- \_\_\_ 4. Send another request with Postman. You should see an error message.
- \_\_\_ 5. Re-start the Bitcoin servers. (Remember to start one on port 9001)
- \_\_\_ 6. Try to reconnect with Postman. The first few tries may fail because it takes time for the services to register with Eureka, then Eureka has to inform Zuul/Ribbon. Eventually the system will return to a *steady-state*.
- \_\_\_ 7. Shut down all servers.
- \_\_\_ 8. Close all.

## Part 7 - Review

In this lab we used Zuul as an edge proxy to route requests, with Ribbon and Eureka to multiple services. We configured *pre* and *post* filters in Zuul and we demonstrated that the system dynamically accounted for server health availability.



## Lab 15 - Distributed tracing with Zipkin

In this lab we will configure Zipkin to trace service requests. We have two services:

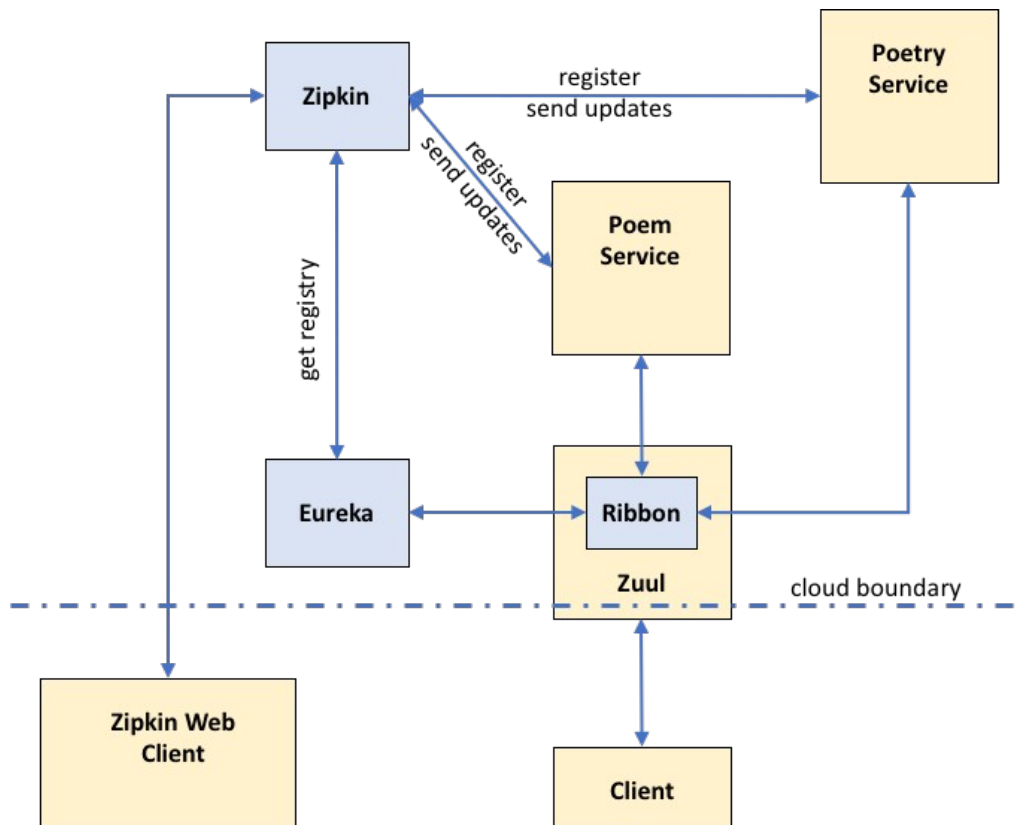
Poem service: has methods that return individual parts of "Fire and Ice" by Robert Frost

Poetry service: makes multiple calls to Poem service to assemble complete poem

All request (including *intra-service* requests) go through Zuul. The services register with Zipkin and send information about requests. We will use the Zipkin Web Client to examine the performance characteristics of the system.

We will make some performance enhancements based on the Zipkin data. Then use Zipkin to evaluate how effective the enhancements are.

This is a high-level diagram of what we will be building in this lab. (Some interactions are not shown for simplicity.)



## Part 1 - Setup

In this part we will perform the basic setup necessary to complete this lab. The core services, Eureka (eureka-service) and Zuul (eureka-service) are the essentially the same as in the Eureka and Zuul labs.

- \_\_\_ 1. Make sure that no Spring Boot applications are running.
- \_\_\_ 2. Examine the C:\LabFiles\Spring-Cloud-Zipkin directory, you should see three directories: eureka-service, gateway-service, poem-service, poetry-service and zipkin-service. These directories contain the partially completed Spring Boot applications that we will use in this lab.

NOTE: for simplicity in this lab we will use the conventions

Shortcut	Path
<eureka-service>	C:\LabFiles\Spring-Cloud-Zipkin\eureka-service
<gateway-service>	C:\LabFiles\Spring-Cloud-Zipkin\gateway-service
<poem-service>	C:\LabFiles\Spring-Cloud-Zipkin\poem-service
<poetry-service>	C:\LabFiles\Spring-Cloud-Zipkin\poetry-service
<zipkin-service>	C:\LabFiles\Spring-Cloud-Zipkin\zipkin-service

## Part 2 - The Core Services

In this part we will start the Zuul and Eureka services.

- \_\_\_ 1. Open a DOS shell to <eureka-service>

\_\_2. Start Eureka by typing the following command in the shell

```
mvn spring-boot:run
```

\_\_3. Open a DOS shell to <gateway-service>

\_\_4. Start Eureka by typing the following command in the shell

```
mvn spring-boot:run
```

### Part 3 - The Poem Service

The **poem-service** contains multiple methods to return the parts of Robert Frost's poem *Fire and Ice*. Here is the complete poem:

Fire and Ice

Some say the world will end in fire,  
Some say in ice.  
From what I've tasted of desire  
I hold with those who favor fire.  
But if it had to perish twice,  
I think I know enough of hate  
To say that for destruction ice  
Is also great  
And would suffice.

by Robert Frost

\_\_1. Open <poem-service>\src\main\resources\bootstrap.properties in a text editor or IDE.

\_\_2. Take note of the service name and port

Service name: \_\_\_\_\_

Port: \_\_\_\_\_

\_\_3. Open <poem-service>\src\main\java\poem\PoemService.java in a text editor or IDE.

\_\_4. Take a look at the class-level @RequestMapping and write down the URL fragment.

Service URL fragment: \_\_\_\_\_

\_\_5. Examine the **getFirstLine()** method. The method returns the first line (sentence) of *Fire and Ice*.

\_\_6. Take note of the method-level @RequestMapping and write down the URL fragment.

Method URL fragment: \_\_\_\_\_

\_\_7. Add the return statement in the getTitle(), getSecondLine(), getThirdLine() and getAuthor() as shown in bold:

```
public String getTitle() {  
    return "";  
}  
...  
public String getSecondLine() {  
    return "";  
}  
...  
public String getThirdLine() {  
    return "";  
}  
...  
public String getAuthor() {  
    return "";  
}
```

\_\_8. Save the file.

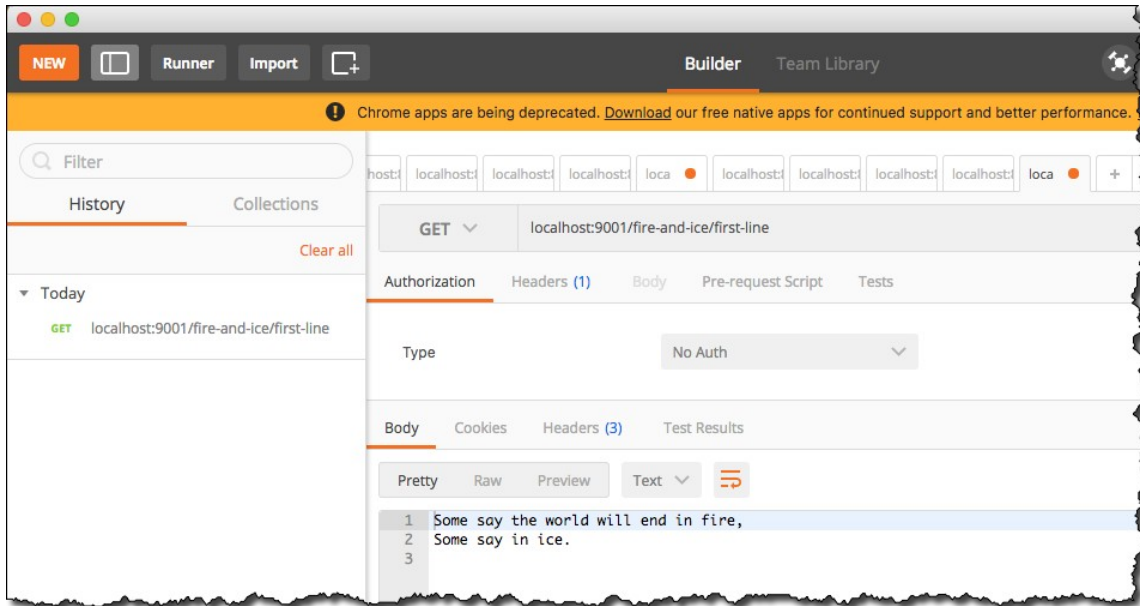
\_\_9. Open a DOS shell to <poem-service>

\_\_10. Start the **poem-service** with the following command:

```
mvn spring-boot:run
```

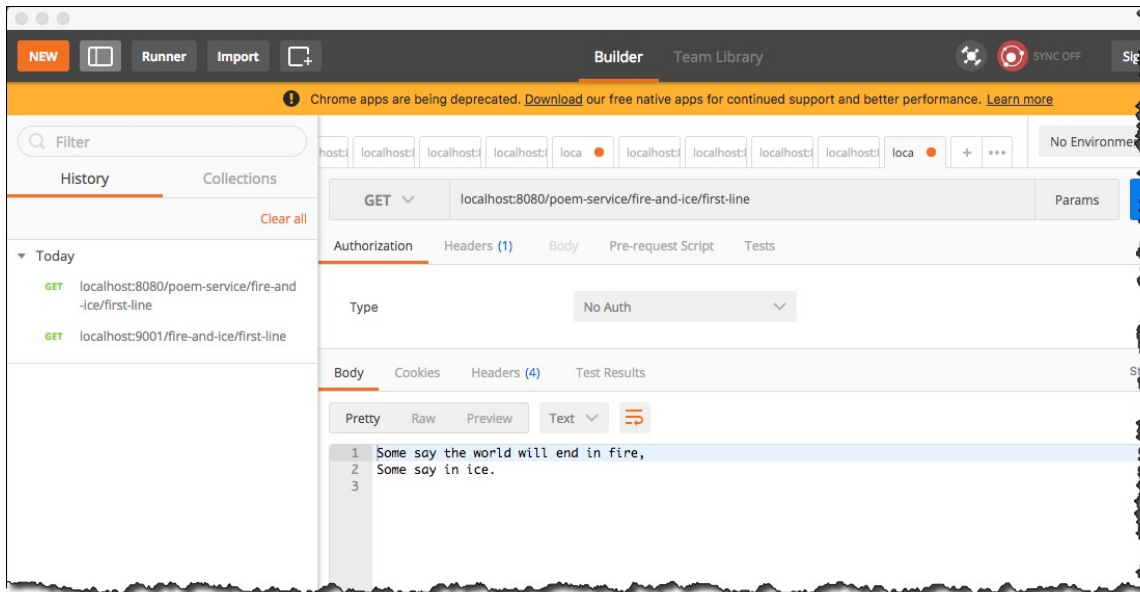
\_\_11. Use Postman to access the **poem-service** directly (replace the values):

`http://localhost:<port>/<service-url-fragment>/<method-url-fragment>`



\_\_12. The **poem-service** is registered with Eureka and Zuul. Use Postman to access the **poem-service** through the Zuul gateway

```
http://localhost:8080/<service-name>/<service-url-fragment>/<method-url-fragment>
```



\_\_13. Complete the getTitle() method to log the title and return it.

```
String title = "Fire and Ice\n\n";

delay(Delay.TIME);

log.info(title);
return title;
```

\_\_14. Complete the getSecondLine() method to log the second line and return it.

```
StringBuilder sb = new StringBuilder();
sb.append("From what Iâ€™ve tasted of desire\n");
sb.append("I hold with those who favor fire.\n");
String secondLine = sb.toString();

delay(Delay.TIME);

log.info(secondLine);
return secondLine;
```

\_\_15. Complete the **getThirdLine()** method to log the third line and return it.

```
StringBuilder sb = new StringBuilder();
sb.append("But if it had to perish twice,\n");
sb.append("I think I know enough of hate\n");
sb.append("To say that for destruction ice\n");
sb.append("Is also great\n");
sb.append("And would suffice.\n");
String thirdLine = sb.toString();

delay(Delay.TIME);

log.info(thirdLine);
return thirdLine;
```

\_\_16. Complete the **getAuthor()** method to log the author's name and return it.

```
String author = "\nby Robert Frost\n";

delay(Delay.TIME);

log.info(author);
return author;
```

\_\_17. Save the file.

\_\_18. Restart the service.

\_\_19. Test again using first-line, second-line, third-line, author and title.

## Part 4 - The Poetry Service 2

The Poetry service has a method that makes multiple calls into the poem service to retrieve the entire poem.

\_\_1. Open **<poetry-service>\src\main\resources\bootstrap.properties** in a text editor or IDE

\_\_2. Take note of the service name and port

Service name: \_\_\_\_\_

Port: \_\_\_\_\_

\_\_3. Open **<poem-service>\src\main\java\poetry\PoetryService.java** in a text editor or IDE

- \_\_4. Examine the **getPoem()** method. Rather than retrieve the entire poem, the initial version of method only gets the title.
- \_\_5. Take note of the method-level **@RequestMapping** and write down the URL fragment

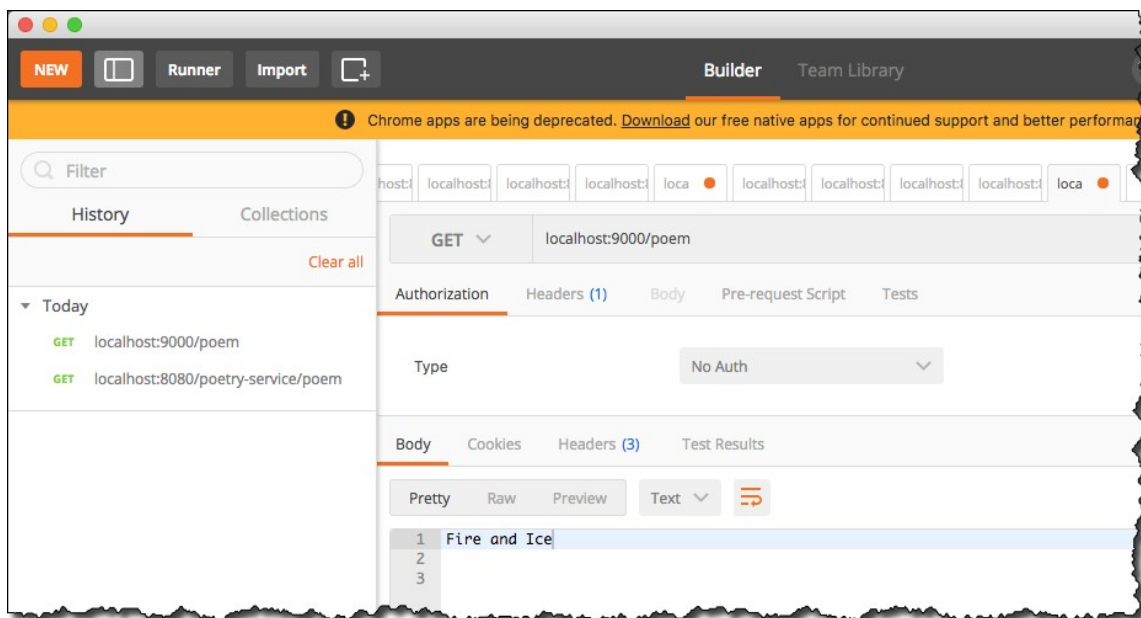
Method URL fragment: \_\_\_\_\_

- \_\_6. Open a DOS shell to **<poetry-service>**
- \_\_7. Start the **poetry-service** with the following command:

```
mvn spring-boot:run
```

- \_\_8. Use Postman to access the **poetry-service** directly

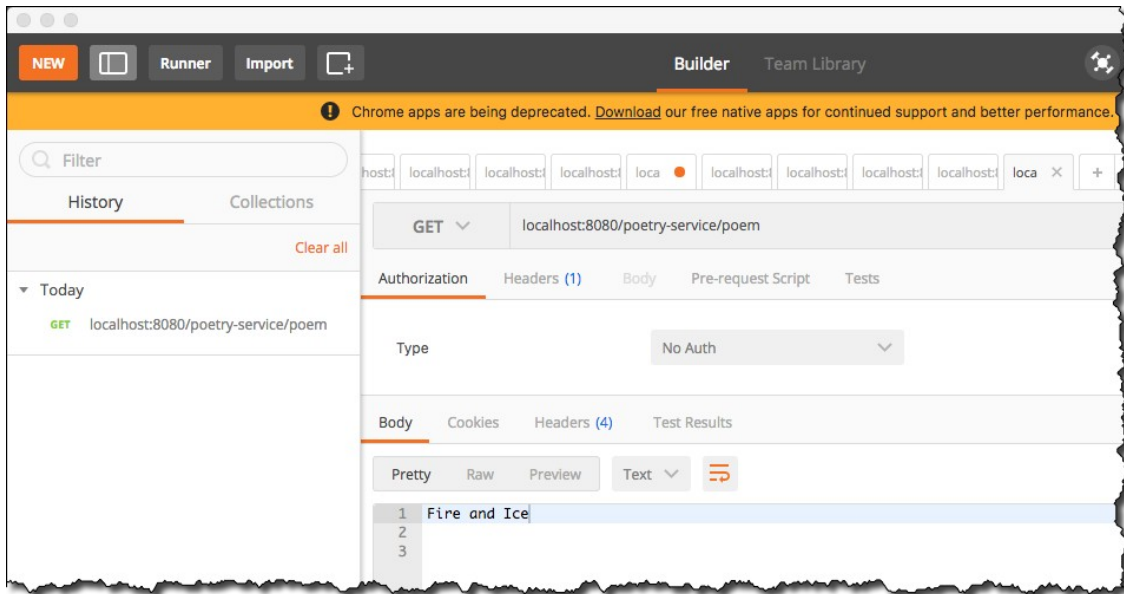
```
http://localhost:9000/method-url-fragment
```



- \_\_9. Use Postman to access the **poetry-service** through Zuul

```
http://localhost:8080/service-name/method-url-fragment
```



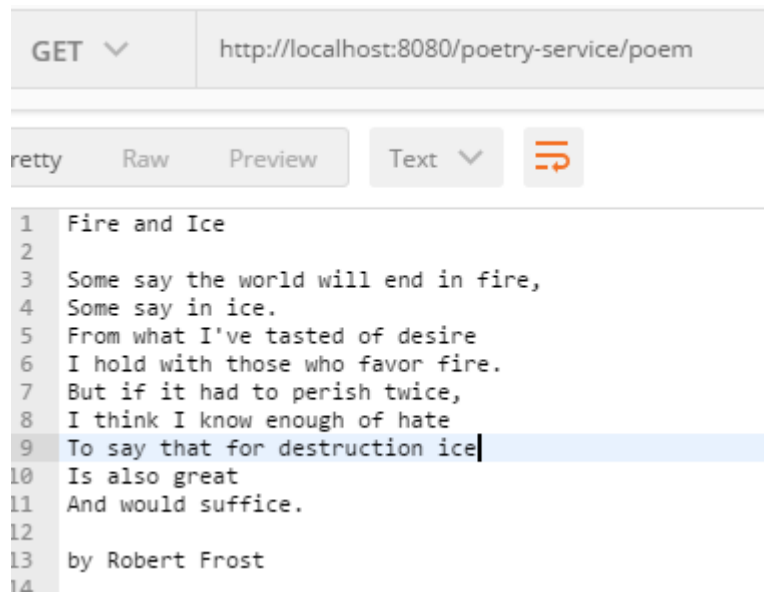


\_\_10. Follow the pattern and modify the **getPoem()** method to retrieve the rest of the poem.

```
sb.append(restTemplate.getForObject(  
    ZUUL_URL+FIRE_AND_ICE+"first-line",  
    String.class));  
  
sb.append(restTemplate.getForObject(  
    ZUUL_URL+FIRE_AND_ICE+"second-line",  
    String.class));  
  
sb.append(restTemplate.getForObject(  
    ZUUL_URL+FIRE_AND_ICE+"third-line",  
    String.class));  
  
sb.append(restTemplate.getForObject(  
    ZUUL_URL+FIRE_AND_ICE+"author",  
    String.class));
```

\_\_11. Save the file.

\_\_12. Restart the **poetry service** and try to get the poem again.



## Part 5 - Configure the Zipkin Service

In this part we will configure and start the Zipkin service to monitor the services in our system.

\_\_1. Open **<zipkin-service>\pom.xml** in a text editor or IDE. Note the Zipkin dependencies.

```
<dependency>
  <groupId>io.zipkin.java</groupId>
  <artifactId>zipkin-server</artifactId>
</dependency>
<dependency>
  <groupId>io.zipkin.java</groupId>
  <artifactId>zipkin-autoconfigure-ui</artifactId>
</dependency>
```

\_\_2. Open **<zipkin-service>\src\main\java\webage\ZipkinService.java**

\_\_3. Add the following annotations to support Zipkin

```
@EnableZipkinServer
@EnableDiscoveryClient
```

\_\_4. Save the file.

\_\_5. Open **<zipkin-service>\src\main\resources\zipkin.properties**

\_\_6. Add the following line to enable registry refresh

```
eureka.client.registryFetchIntervalSeconds=5
```

\_\_7. Save the file.

\_\_8. Open a DOS shell to <zipkin-service> and start the Zipkin service with the following command

```
mvn spring-boot:run
```

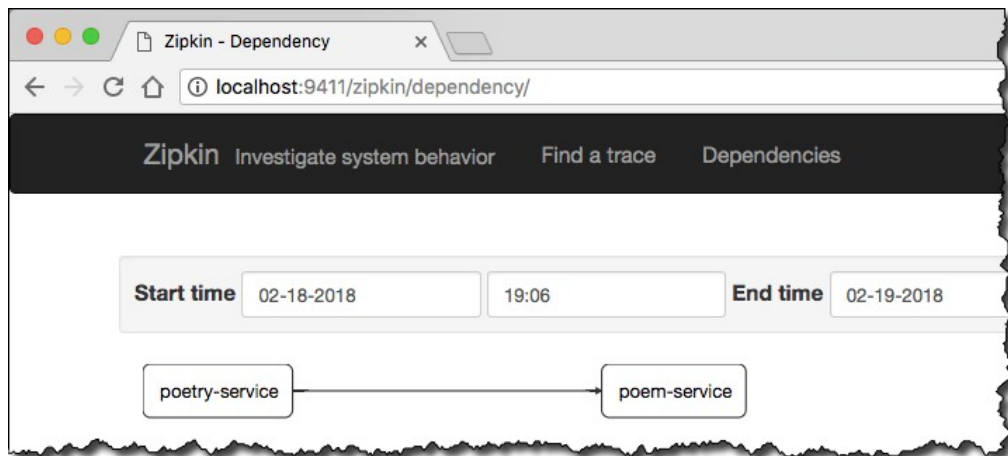
## Part 6 - Monitor the System with Zipkin

In this part we will use Zipkin to examine the services in our system. Then we will use the information to improve system performance.

\_\_1. Open the Zipkin console in a browser at **http://localhost:9411**

\_\_2. Use Postman to make some requests to the poetry-service.

\_\_3. Select **Dependencies** and view the service dependencies. As expected, the poetry-service depends on the poem-service.



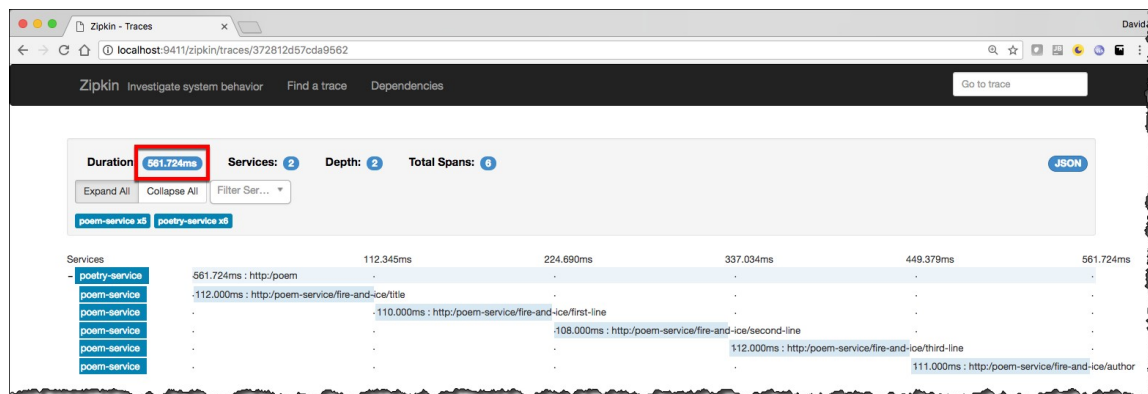
\_\_4. Go back to the Zipkin home page.

\_\_5. Click on **Find Trace** (In this case we are not filling in any of the search criteria)

\_\_6. Clear out the value in the time field of the start time.

Service Name: all  
Span Name: E  
Start time: 02-16-2018

\_\_7. Select one of the traces. Notice the total time for the request. Also notice the five calls to the **poem-service** are executed serially. Since each method has a 100ms delay the total time is (5x100ms + extra).



## Part 7 - Optimize the Solution

The current solution is not scalable. For a large poem with multiple stanzas we would have to make several calls to the poem service. The time taken for the poetry-service increases linearly with the number of calls to the poem-service. Adding concurrency to the poetry-service should make it more scalable.

We will use the **CompletableFuture** class and a **TaskExecutor** to perform the requests in parallel.

This code uses the **Scatter-Gather** pattern. Multiple requests are made in parallel (scatter) and when the results are all complete they are collated (gathered) and returned.

\_\_1. Open `<poetry-service>\src\main\java\poetry\PoetryService.java` in a text editor or IDE.

\_\_2. Create an new class in the **PoetryService.java** file to define a **TaskExecutor**. The class can't be public and should be a top-level class. Set the thread pool size for the executor to six.

```
@Configuration
class ThreadConfig {
    @Bean
    public TaskExecutor threadPoolTaskExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(6);
        executor.setMaxPoolSize(6);
        executor.setThreadNamePrefix("default_task_executor_thread");
        executor.initialize();
        return executor;
    }
}
```

\_\_3. Add a **TaskExecutor** variable to the **PoetryService** class

```
@Autowired
private TaskExecutor taskExecutor;
```

\_\_4. Add a function yo the **PoetryService** class to retrieve sections of the poem asynchronously

```
@Async
public String getPoemSection(String sectionName) {
    String url = ZUUL_URL+FIRE_AND_ICE+sectionName;
    String section = restTemplate.getForObject(url, String.class);
    return section;
}
```

\_\_5. Add a function to retrieve the entire poem mapped to the URL pattern /poem-async

```
@RequestMapping("/poem-async")
public String getPoemAsync() throws ExecutionException,
InterruptedException {

    CompletableFuture<String> title =
        CompletableFuture.supplyAsync(
            ()->getPoemSection("title"),taskExecutor);
    CompletableFuture<String> firstLine =
        CompletableFuture.supplyAsync(
            ()->getPoemSection("first-line"),taskExecutor);
    CompletableFuture<String> secondLine =
        CompletableFuture.supplyAsync(
            ()->getPoemSection("second-line"),taskExecutor);
```

```

CompletableFuture<String> thirdLine =
    CompletableFuture.supplyAsync(
        ()->getPoemSection("third-line"),taskExecutor);
CompletableFuture<String> author =
    CompletableFuture.supplyAsync(
        ()->getPoemSection("author"),taskExecutor);

// wait for all results
CompletableFuture.allOf(
    title,
    firstLine,
    secondLine,
    thirdLine,
    author).join();

String poem = title.get() +
    firstLine.get() +
    secondLine.get() +
    thirdLine.get() +
    author.get();

log.info(poem);
return poem;
}

```

\_\_6. Save the file.

\_\_7. Open the DOS shell to **<poetry-service>**

\_\_8. Restart the **poetry-service** by entering the following command

```
mvn spring-boot:run
```

## Part 8 - Evaluate the Solution with Zipkin

In this part we will use Zipkin to evaluate the updated solution.

\_\_1. Open the Zipkin console in a browser at **http://localhost:9411**

\_\_2. Use Postman to make some requests to new method in the poetry-service at the following url.

```
http://localhost:8080/poetry-service/poem-async
```

\_\_3. Go back to the Zipkin home page.

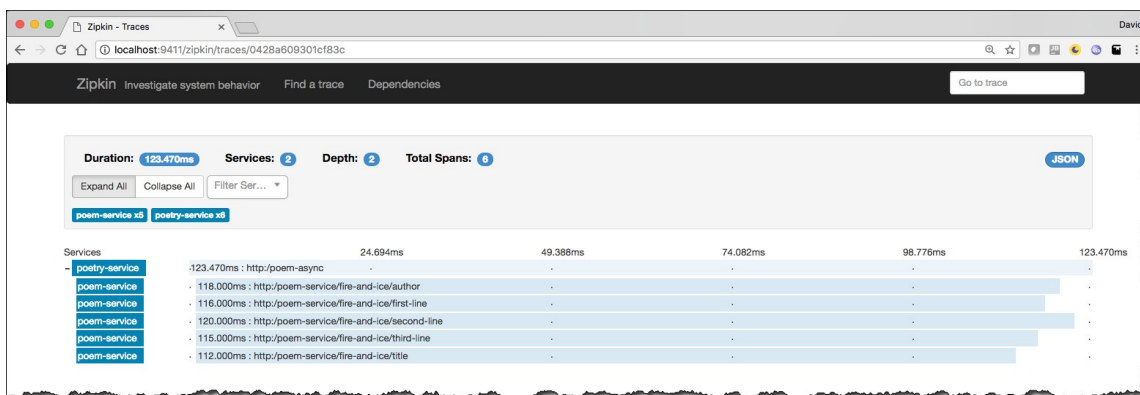
\_\_4. **Click on Find Traces** (clear out the start time or enter the closest time).

\_\_\_5. Select a recent trace. You may need to wait some minutes to see the trace.



Troubleshoot, if you don't see the trace then restart all services in the following order and then do some requests: eureka-service, gateway-service, poem-service, poetry-service and zipkin-service.

\_\_\_6. Review the trace. Notice that this time the calls are made in parallel so the total time is now (100ms + extra).



\_\_\_7. Stop all services and close all.

## Part 9 - Review

In this lab we used Zipkin to monitor services. We used the data to optimize our services and verify that the optimizations were effective.

## Lab 16 - Spring Boot Project

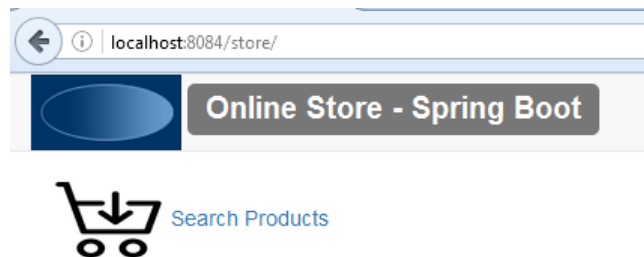
This document will describe the general requirements of a Spring Boot “project”. This project is meant to supply limited details on how to accomplish tasks and test your ability to implement the project requirements using Spring Boot based on the other experience you have received during Spring Boot training.

### Part 1 - Project Requirements

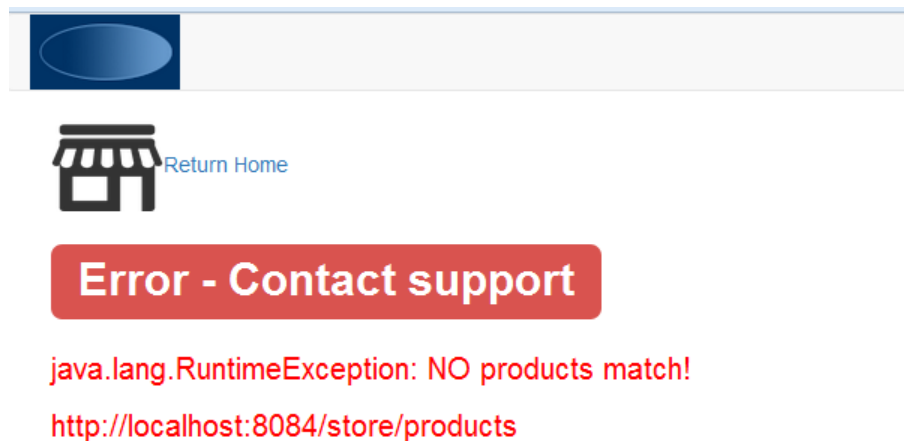
The goal of the project is to implement a web application that allows to search and display product information in an “Online Store”. Various Spring and Spring MVC components will be used to access this data from a Spring Data repository. A Spring Boot configuration will automatically configure a number of features used by the application.

The following pages will be implemented as part of the project:

#### HOME PAGE



#### ERROR PAGE






## INITIAL PRODUCT LIST PAGE

localhost:8084/store/products

Online Store - Product List

 [Return Home](#)

Product Category:

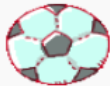

Max Price: (Zero for no maximum)

Image	Id	Name	Price	Categ
-------	----	------	-------	-------

## PRODUCT LIST AFTER A SEARCH

Product Category:

Max Price: (Zero for no maximum)

Image	Id	Name	Price	Category	Description
	4	Soccer Ball	19.99	Sports	This youth size soccer ball is g and comes with an inflation ne
	7	Skateboard	19.99	Sports	This rugged skateboard is toug comes with tools for adjusting t

## Part 2 - Project Data Structure

**PRODUCT:** Product table

Column Name	Column Type	Column Description
PRODUCTID	BIGINT NOT NULL	Product ID number, internally generated. This is a primary key.
NAME	VARCHAR(20) NOT NULL	Product name.
CATEGORY	VARCHAR(20) NOT NULL	Product category. Will be used in search.
DESCRIPTION	VARCHAR(250) NOT NULL	Product description.
IMAGEURI	VARCHAR(40) NOT NULL	File to be used for an image of the product.
PRICE	DECIMAL(10,2)	Product price.

## Part 3 - Project Setup

The following steps describe general steps to take to configure the Eclipse project to support the project development.

The following general software requirements from the Spring Boot training are required:

- Java 8 JDK with 'JAVA\_HOME' and 'PATH' variables adjusted appropriately
- Eclipse for Java EE Developers – Mars edition

\_\_1. Create a new Eclipse project called 'OnlineStore'. Make sure the following things are true about the project:

- This should be a Maven project but skip the Maven archetype selection to create a simple project.
- Use the 'spring-boot-starter-parent' Spring Boot artifact as the parent of the project
- Use 'jar' packaging
- Make sure the project is linked to the Java 8 runtime for the build path

\_\_2. Add the following dependencies to the Maven pom.xml configuration:

- spring-boot-starter-web (exclude 'spring-boot-starter-logging' from this dependency)
- spring-boot-starter-actuator
- spring-boot-starter-log4j
- spring-boot-starter-thymeleaf
- spring-boot-starter-jdbc
- spring-boot-starter-data-jpa
- spring-boot-starter-data-rest
- h2 (database as a runtime dependency)

## **Part 4 - Create Project Configuration Files**

Various files in a Spring Boot application can configure various services and utilities. In this part some of the basic files will be described for you to add.

\_\_1. Create a 'log4j.properties' file

- Located in the 'src/main/resources' project folder
- Configure the 'rootLogger' to show INFO messages and use a file appender
- Configure the file appender to write to a 'store.log' file and disable appending to the file
- Configure the file appender to use a PatternLayout with a reasonable 'ConversionPattern' output

\_\_2. Create the following empty folders in the 'src/main/resources' project folder

- config
- static
- templates

\_\_3. Create a 'BootApplication' Java class that will run the Spring Boot Application

- Create the class in the 'com.webage' package in the 'src/main/java' folder of the project
- Call the class 'BootApplication'
- Add the '@SpringBootApplication' annotation to the class
- Create a main method that creates an instance of the 'SpringApplication' class and calls the 'run' method

\_\_4. Create a 'schema.sql' file that will create the Product table required for the application data.

- Located in the 'src/main/resources' project folder
- Use the description of the data structure from the earlier lab section

\_\_5. Create a 'data.sql' file that will insert the sample data used by the application.

- Located in the 'src/main/resources' project folder
- Use the 'products.txt' file from the lab files for the project. This is a comma-separated list of the values used for the data.

\_\_6. Create an 'application.properties' file with various application configuration properties:

- Located in the 'src/main/resources/config' project folder
- Disable Hibernate and JPA database creation since our schema.sql and data.sql files will do that.
- Set the server port to 8084
- Set the server contextPath to '/store'

## **Part 5 - Design and Test Static Resources**

To help verify configuration of the project and start defining the pages that will be part of the application, this section will describe how to define some static resources and test they are available.

\_\_1. Create an empty 'home.html' file in the 'templates' folder (this is required to avoid a Thymeleaf startup error)

\_\_2. Copy static resources

- Copy the 'static' subfolder of the lab files for the project into the 'src/main/resources/static' project folder
- To check the correct placement, check you have a 'src/main/resources/static/images' project folder with various image files

\_\_3. Run the 'BootApplication' class as a Java class and check project configuration

- The Java process should stay running and not exit because of any errors.
- The store.log file should be created and have various messages about the successful startup of the Apring Boot application
- Various URLs like the following should display the static image files in the project

<http://localhost:8084/store/images/a003cut.gif>

\_\_\_4. Terminate the Java process in the Console view. Fix any errors and retest until the tests above pass. Make sure to close the store.log file if you opened it.

\_\_\_5. Open the 'application.properties' file and add properties for the application title and properties for the title of the home page and product list page that uses the application title property

\_\_\_6. Create a Java class that will act as a Controller for the home page of the application

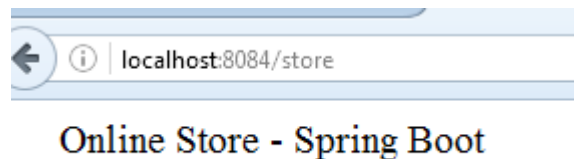
- It should have any appropriate Spring MVC annotations.
- It should contain a method with a request mapping to the “root” ("/) of the application
- Obtain the value of the home page title property and add it as an attribute of the Spring MVC Model to be available in the view page
- Return “home” to direct to the 'home.html' view.

\_\_\_7. Open the 'home.html' file and add some simple code to use Thymeleaf to display the text of the title attribute of the Model. This will check it is being set correctly.

\_\_\_8. Run the 'BootApplication' class to start the application.

\_\_\_9. Open the following address in a browser to check that the page is displayed.

<http://localhost:8084/store>



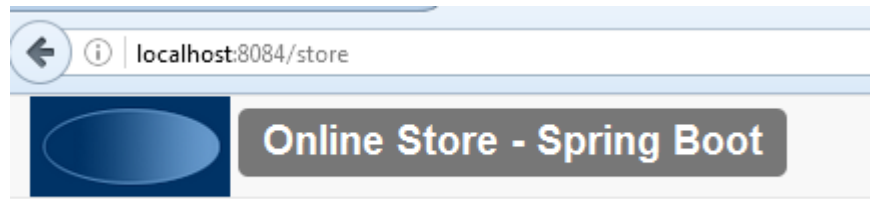
\_\_\_10. Terminate the Java process.

\_\_\_11. Create a 'header.html' file that will include various aspects reused by pages in the application

- Declaration of a Thymeleaf 'header' fragment to be referenced from other pages
- Link to the Bootstrap stylesheet included in the static resources
- Inclusion of the '/images/theme/logo\_blue.gif' image file
- Include the title text from the Model. This will be removed from the 'home.html' page.

\_\_\_12. Open the 'home.html' page and modify it to remove the display of the title property and instead include the header fragment using a Thymeleaf 'replace' attribute.

\_\_13. Rerun the application and open the home page again. This time a more stylized page should be displayed with information from the header.



\_\_14. Terminate the Java process

## Part 6 - Define JPA Entity Class

The next step will be to begin building up the code involved in retrieving and working with the product data from the database. The first step in this can be to create a JPA Entity class to model the data in Java.

- \_\_1. Create a new 'Product' Java class in a package in your application.
- \_\_2. Add fields to the Java class that match the data stored in the database table and use a Java type that makes sense for the field.
- \_\_3. Add or generate “getter/setter” methods for the fields of the class.
- \_\_4. Add the JPA '@Entity' annotation to the class.
- \_\_5. Add the appropriate JPA annotations to identify the field used as the primary key and that the value is generated automatically by the database.
- \_\_6. If required, add the JPA annotations to map the Java class to the correct table and the fields to the appropriate column in the table.
- \_\_7. Add a 'toString' method to return a reasonable String representation of the data of the object.

## Part 7 - Create a Data Repository

The various Spring Data projects can expose a data repository automatically generated from the JPA data of the application. In this section you will create the interface to enable this.

- \_\_1. Create a Java interface 'ProductRepository' in a package.
- \_\_2. Extend the Spring Data 'CrudRepository' interface and indicate that the repository works with 'Product' data with 'Long' values as primary keys.

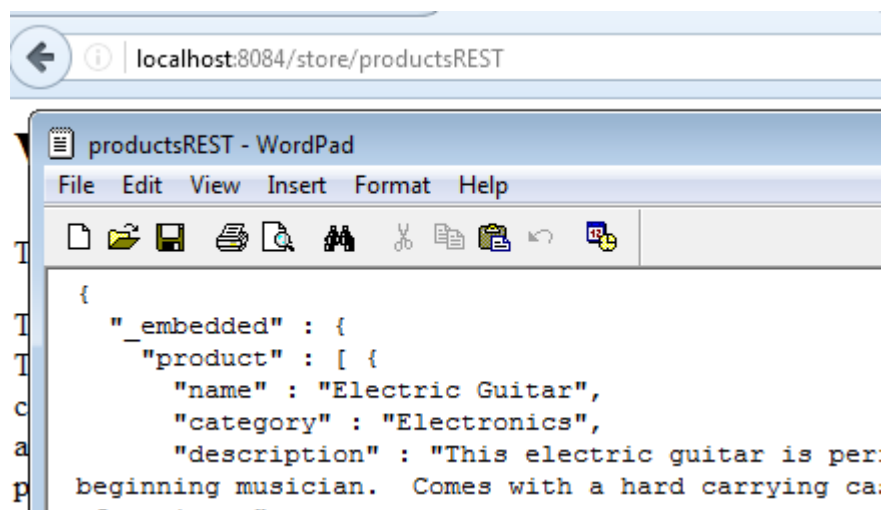
\_\_3. Add the '@RepositoryRestResource' Spring Data annotation with appropriate annotation property values to the interface.

\_\_4. Add the three appropriate methods to the interface to allow retrieving matching Product information by one of the following criteria:

- Category
- Category and Price less than a value
- Price less than a value

\_\_5. Run the application again and use a browser to open the URL below, substituting your value of the 'path' attribute of the 'RepositoryRestResource' annotation. You will likely be prompted to open or save the response but should eventually be able to see the data which will prove the data can be retrieved from the database and made available through the repository.

<http://localhost:8084/store/{path}>



## Part 8 - Define Spring Service

Since there will be searches for product information, it will be good to perform these with a Spring service component. This service component can have some logic for calling the appropriate method from the data repository based on the search parameters supplied.

\_\_1. Define a new 'ProductService' interface in a Java package.

\_\_2. Add the following methods to the service interface:

- A 'find' method that takes a productID and returns a single product
- A 'productSearch' method that takes a String for the category and a double for the maximum price and returns a Collection of matching Products

- \_\_3. Add a new Java class that implements this service interface.
- \_\_4. Mark the Java class that implements the service interface with the appropriate annotation to declare it as a Spring component.
- \_\_5. Add a field to the class to store a reference to the data repository and add the appropriate Spring annotation to inject the repository component.
- \_\_6. Modify the implementation of the 'find' method to use the appropriate method of the data repository to locate a single Product by the ID.
- \_\_7. Modify the implementation of the 'productSearch' method to use the various methods of the data repository to return appropriate search results based on various conditions as described below.
  - Category of “ALL” and zero or negative maximum price → return all products
  - Category of “ALL” and positive maximum price → return products based on price only
  - Category besides “ALL” and zero or negative maximum price → return products based on category only
  - Category besides “ALL” and positive maximum price → return products based on category and price

## **Part 9 - Add Product List Controller**

The next step will be to come up with a Spring Controller that will handle the product list page. This will include the initial display of the page along with any searches that are performed with the product search form on the page.

- \_\_1. Create a Java class that will act as a Controller for the product list page of the application
  - It should have any appropriate Spring MVC annotations at the class level to declare it as a controller and provide a request mapping to the path that will be used for the product list page. Make sure to use a different path than what is used for the REST repository.
- \_\_2. Add a field and appropriate Spring injection annotation for a reference to the Spring service that can perform product searches.



\_\_3. Create the following Java methods that will construct the appropriate Java object and return it to link it to the given Spring model attribute (using the appropriate Spring annotation).

- title – A String that provides the page title for the product list page.
- subtitle – A String like “Search for Products” to indicate the purpose of the page.
- categoryNames – Returns a List of Strings that are the category names. This should include the “ALL” option as well as all other categories from the product data.

\_\_4. Create a new method in the controller that will have a request mapping to a GET request.

- Have the method set a model attribute for the 'maxPrice' of 0.0. This will be the default on the form.
- Have the method return a String for the name of the product list view.

\_\_5. Create a new method in the controller that will have a request mapping for a POST request. It should take a request parameter for a 'category' and 'maxPrice' along with the Spring MVC Model.

- In the method call the appropriate product search method in the service passing the request parameters for category and maxPrice. Store the return value as a Collection of products.
- Check if the list of products is empty and thrown an Exception with an appropriate message if it is.
- Store the list of products and the value of the 'maxPrice' as Model attributes.
- Return a String for the name of the product list view.

## Part 10 - Implement Product List Page

With the structure of the controller for the page established, you can create the page to list products. This will show a blank form when first displayed and then show the products that match the search when the form is submitted.

\_\_1. Create an empty HTML file in the 'templates' folder for the product list page. Make sure the name of the file matches the view name that is returned by the controller.

\_\_2. Add code for some basic elements of the product list page:

- Link to the header fragment
- Display the provided 'storeicon' image and text like “Return Home” as a link back to the context root of the web application
- The value of the 'subtitle' property text

\_\_3. Modify the 'home.html' file to display the 'shopicon' image file and text like “Search Products” as a link to the product list page.

\_\_4. At this point you should be able to run the application, open the home page in a browser and go between the pages with the links supplied.

\_\_5. Modify the product list page to show a form to search for products and a table to display the results. It should have the following features:

- A form that will submit a POST request back to the same product list path
- A select element for the category selected which is populated by the 'categoryNames' model property from the controller
- A text box for the maximum price to search for
- Description on the maximum price to indicate a zero value will indicate no maximum.
- A table that will iterate over the returned products and display relevant properties. This should include the use of the 'imageURI' property of the product to display the image.

\_\_6. Run the application and make sure you can search for products and they are properly displayed.



**Product Category:**

ALL

**Max Price: (Zero for no maximum)**

0

Submit

Image	Id	Name	Price	Category	Description
	1	Electric Guitar	59.99	Electronics	This electric guitar comes with a case and an amplifier.
	2	Computer	799.99	Electronics	This computer comes with a LCD monitor.

## Part 11 - Add Error Handling

One of the last steps is to improve on the generic error page displayed when an error occurs. This is done with a Spring MVC Exception Handler.

- \_\_1. In the same package as the Spring MVC controllers, create a Java class for the error advice.
- \_\_2. Mark the class with the Spring annotation that will enable the class as a controller advice class.
- \_\_3. Add a method that will take the Servlet request and an Exception as parameters and return a Spring MVC ModelAndView object
- \_\_4. Add the annotation to the method to mark it as a exception handler for all exceptions.
- \_\_5. To the ModelAndView returned by the method, add objects to display on the error page like the URL and the exception, and set an appropriate view name.
- \_\_6. Create an empty HTML file in the 'templates' folder for the error page. Make sure the name of the file matches the view name that is set by the controller advice.
- \_\_7. Add the following features to the error page:
  - Inclusion of the header fragment
  - Display the provided 'storeicon' image and text like “Return Home” as a link back to the context root of the web application
  - Some type of header to indicate the page is meant to display an error
  - Display of the text of the exception and the URL that was being displayed
- \_\_8. Run the application and generate an error to test the error page. One way to do this can be to search for products with a positive maximum price that is lower than all products.



### Error - Contact support

**java.lang.RuntimeException: NO products match!**

**<http://localhost:8084/store/products>**

## **Part 12 - Review**

The requirements and guidance in this project were intended to allow you to implement a Spring Boot web application by using knowledge obtained as part of Spring Boot training. By practicing how to implement the application requirements using Spring Boot with minimal guidance and no direct code provided you can become more familiar with how to use Spring Boot to implement your own web applications.

