

## 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 - Create a RESTful API with Spring Boot .....	31
Lab 6 - Create a RESTful Client with Spring Boot .....	44
Lab 7 - Enable Basic Security.....	48
Lab 8 - Use AMQP Messaging with Spring Boot .....	52

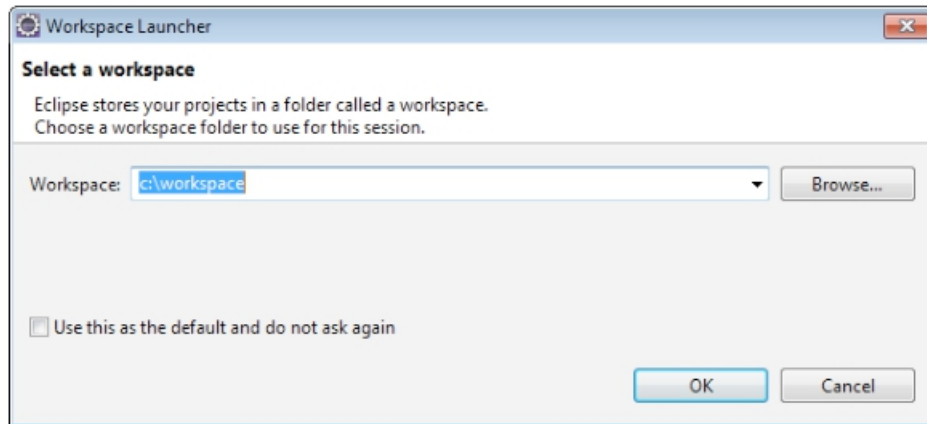
## Lab 1 - A Simple RESTful API in Spring Boot

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** .

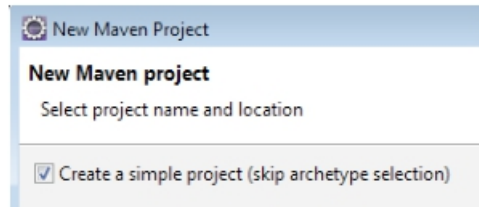


- \_\_3. Close the **Welcome** panel by clicking on the 'X':



- \_\_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.kiddcorp.spring.samples

**ArtifactId:** helloAPI

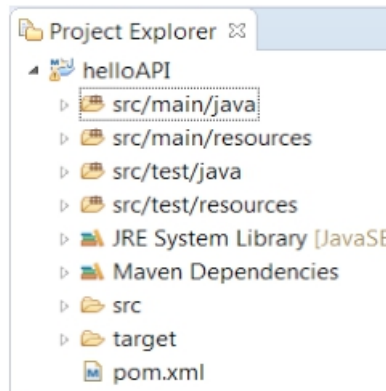
Leave all the other fields at their default values.

\_\_7. Click **Finish** .

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

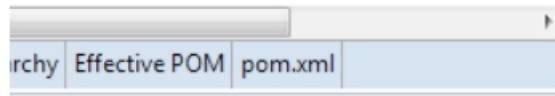
The steps so far have created a basic Maven project. Now we'll add the dependencies to make a Spring Boot project.

\_\_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.kiddcorp.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.

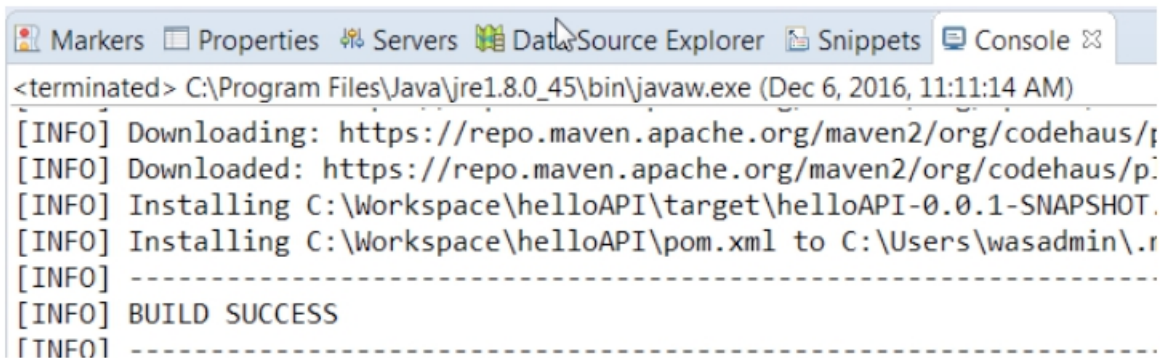
\_\_8. Save the file.

\_\_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/j
[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.kiddcorp.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.kiddcorp.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.kiddcorp.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.

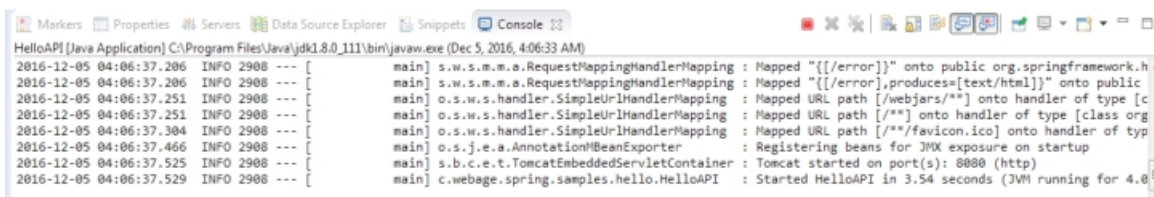
## Part 5 - Run and Test

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:

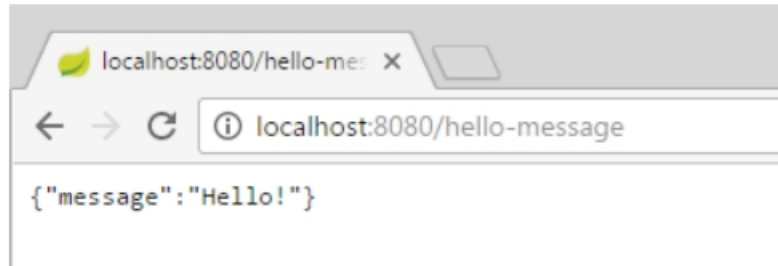


```
2016-12-05 04:06:37.206 INFO 2908 --- [main] s.w.s.m.s.RequestMappingHandlerMapping : Mapped "[]" onto public org.springframework.h  
2016-12-05 04:06:37.206 INFO 2908 --- [main] s.w.s.m.s.RequestMappingHandlerMapping : Mapped "[[/error], produces=[text/html]]" onto public  
2016-12-05 04:06:37.251 INFO 2908 --- [main] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/webjars/**] onto handler of type [c  
2016-12-05 04:06:37.251 INFO 2908 --- [main] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/**] onto handler of type [class org  
2016-12-05 04:06:37.304 INFO 2908 --- [main] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/**/favicon.ico] onto handler of typ  
2016-12-05 04:06:37.466 INFO 2908 --- [main] o.s.j.e.a.AnnotationMBeanExporter : Registering beans for JMX exposure on startup  
2016-12-05 04:06:37.525 INFO 2908 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)  
2016-12-05 04:06:37.529 INFO 2908 --- [main] c.webpage.spring.samples.hello.HelloAPI : Started HelloAPI in 3.54 seconds (JVM running for 4.0
```

\_\_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).

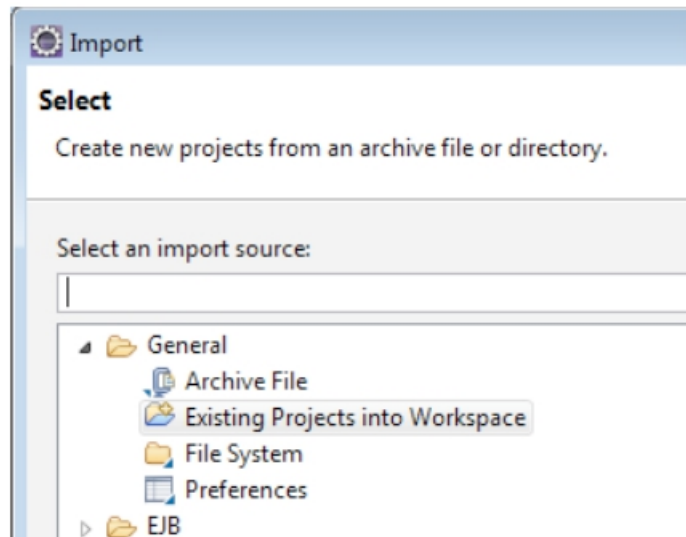
## Lab 2 - Use the Spring Web MVC Web Framework under Spring Boot

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.

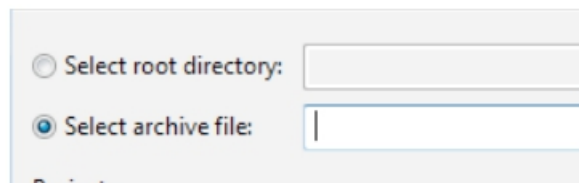
In this lab you will use some of these features of Spring Web MVC in the Spring Boot environment. This will be a simple application that manages "Purchase" data but will be enough to demonstrate the main features of the Spring Web MVC framework.

### Part 1 - Lab Setup

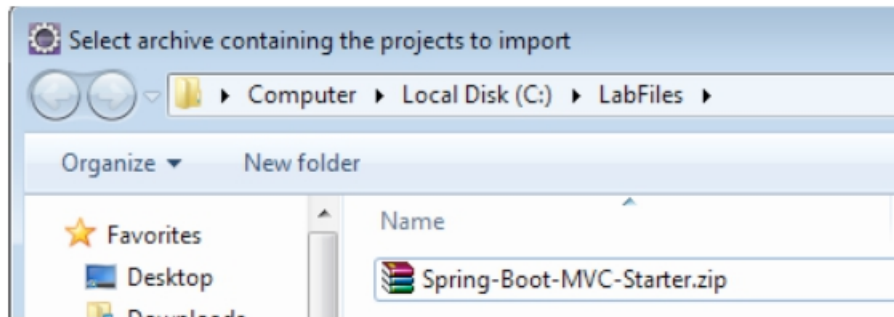
1. From Eclipse's main menu, click **File** → **Import** and then select **General** → **Existing Projects into Workspace** , and then click **Next** .



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

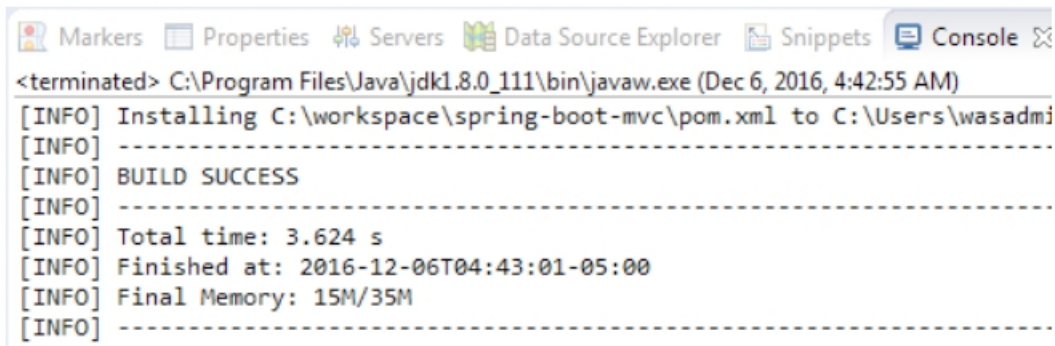
Have a look at the project that we've just imported. In particular, note the following:

- 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.kiddcorp'. 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.kiddcorp.dao' package contains an in-memory implementation of a storage repository for purchases.
- The 'com.kiddcorp.domain' package contains a domain object for purchases.
- The 'com.kiddcorp.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.kiddcorp.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.

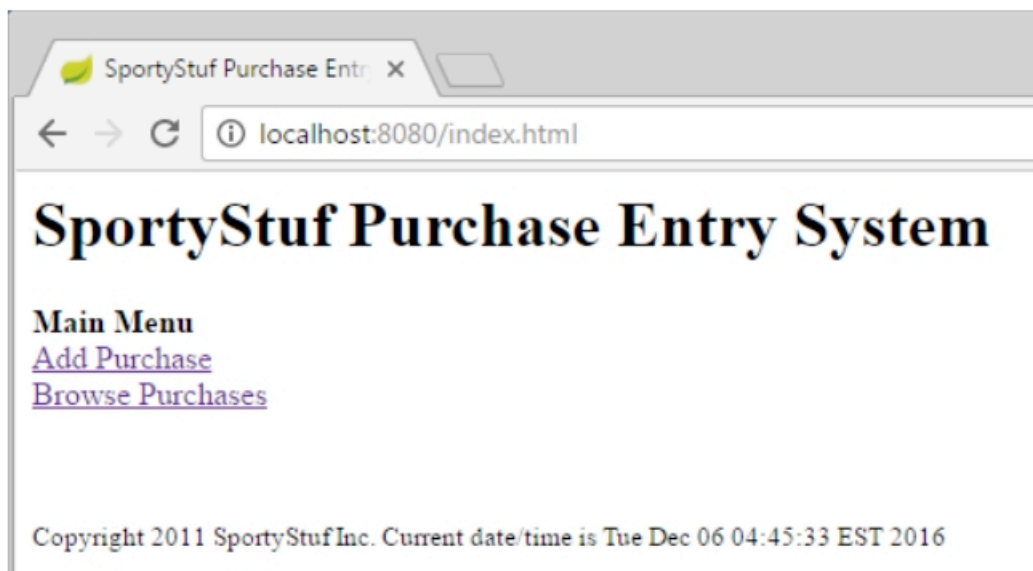
1. In the **Project Explorer** right-click on the **spring-boot-mvc** project and select **Run as** → **Maven install**. The build should run successfully. You may need to build twice since the first time sometimes doesn't build fine.



```
<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.kiddcorp** 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.kiddcorp.web**

\_\_6. Open the **PurchaseController.java** file in the **com.kiddcorp.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](#)

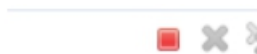
	Purchase Id	Customer Name	Purchase Date	Product
Edit	1	Susan	May 12, 2010	Mountain Bike
Edit	2	Bob	Apr 30, 2010	Football
Edit	3	Jill	Jun 5, 2010	Kayak

[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.



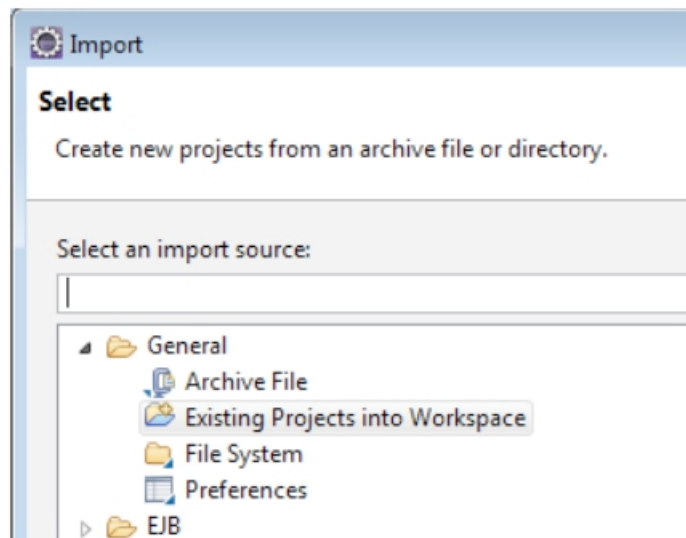
## Lab 3 - Use the Spring JDBCTemplate under Spring Boot

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.

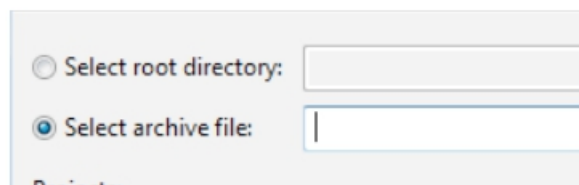
In this lab you will implement a Data Access Object by creating SQL queries and issuing them with a JDBCTemplate that is autowired by Spring.

### 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** .

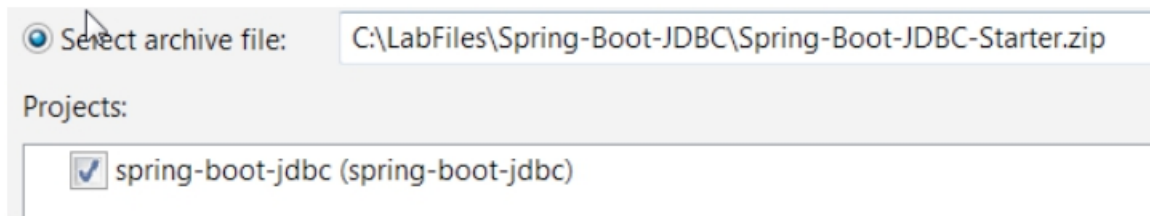


\_\_2. Click the radio button for **Select archive file**:



\_\_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.kiddcorp.dao' in the **Project Explorer** under '**spring-boot-jdbc/src/main/java**'.

\_\_2. Right-click on the 'com.kiddcorp.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.kiddcorp.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
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

- \_\_5.
- \_\_6.
- \_\_7.

ation:



## **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.

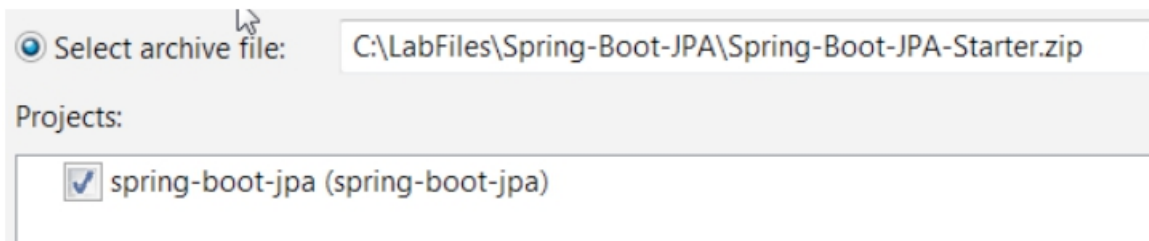
## Lab 4 - Use the Spring Data JPA under Spring Boot

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.kiddcorp.domain' in the **Project Explorer**.

\_\_2. Locate the **Customer** class inside 'com.kiddcorp.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.kiddcorp.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.kiddcorp.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.kiddcorp.repository' package. Double-click on the class to open it.

For convenience, the contents are reproduced below:

```
package com.kiddcorp.repository;

import org.springframework.data.repository.CrudRepository;

import com.kiddcorp.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.kiddcorp.services' package. Double-click on the class to open it. For convenience, the contents are shown below:

```
package com.kiddcorp.service;

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

import com.kiddcorp.domain.Purchase;
import com.kiddcorp.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

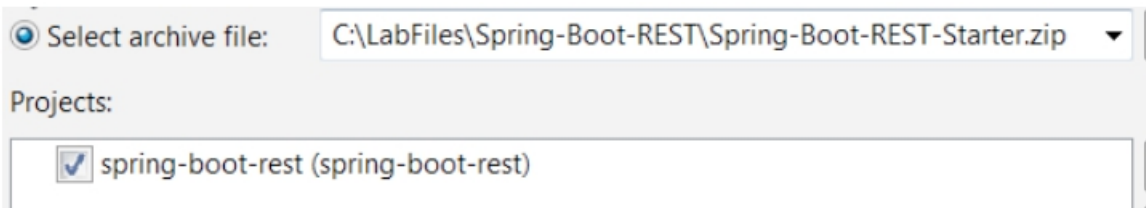
In this lab, we used Spring Data's functionality combined with Spring JPA to implement a data repository very rapidly, with a minimum of effort.

## Lab 5 - 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.kiddcorp.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'.

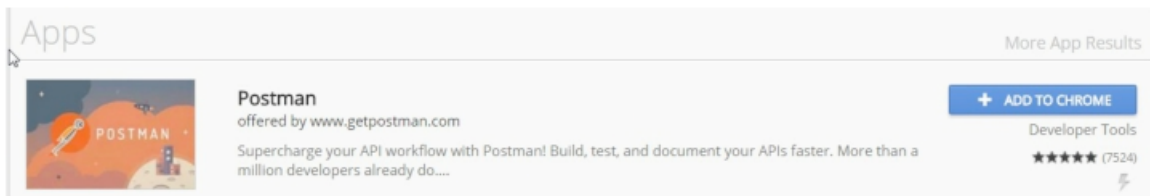
\_\_2. Run the 'App' class as a **Java Application** .

The system should start up without errors.

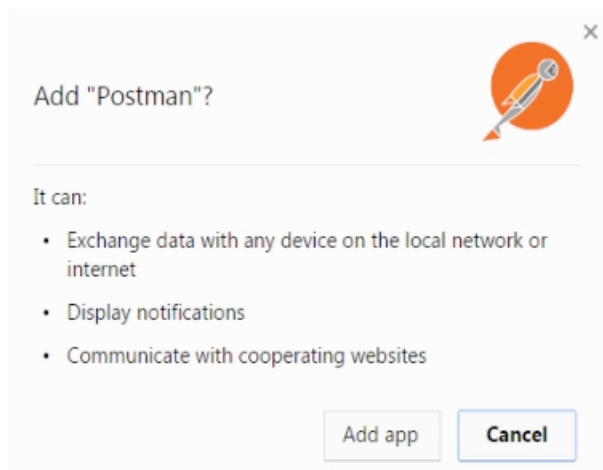
\_\_3. Open Chrome and find the Postman App.

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

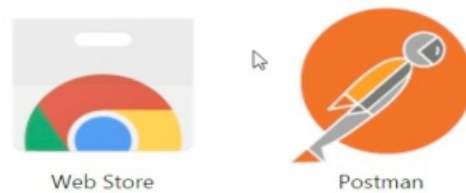
\_\_4. Select the **Postman** under **Apps** and click **ADD TO CHROME** .



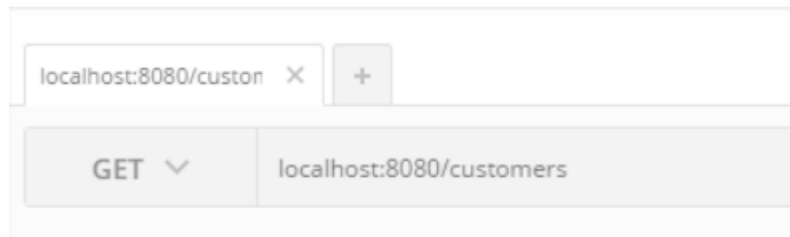
\_\_5. Click **Add App** .



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

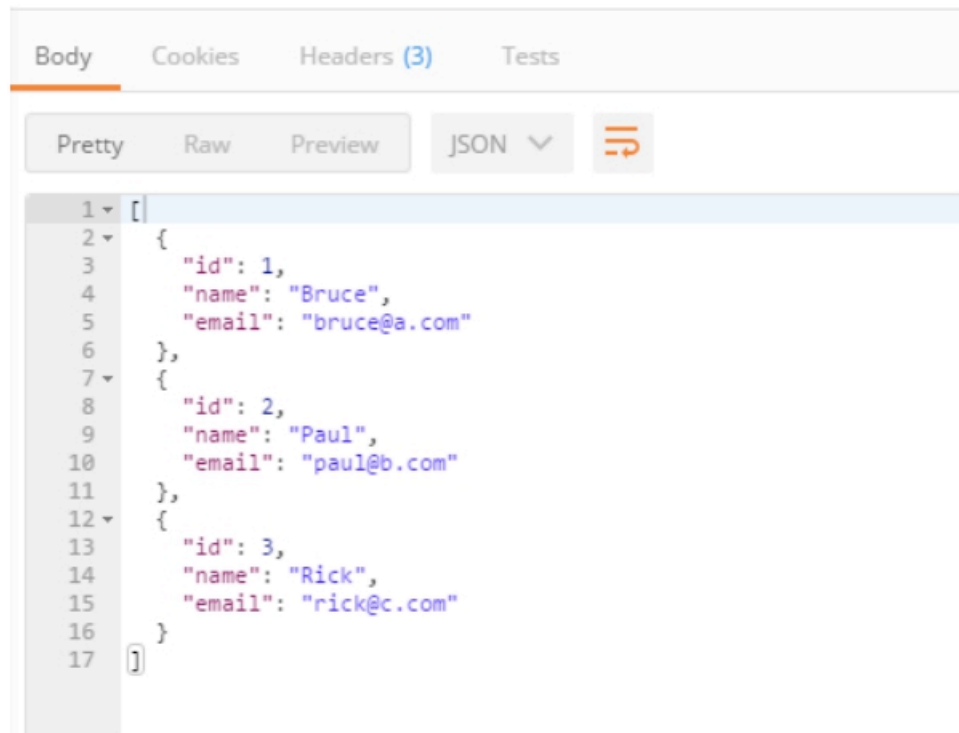


7. Select the a 'GET' request using the drop-down box, and enter 'localhost:8080/customers' as the URL:



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.

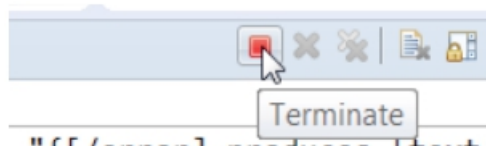
\_\_3. Annotate the 'long id' parameter as follows:

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

\_\_4. Organize imports.

\_\_5. Save the file.

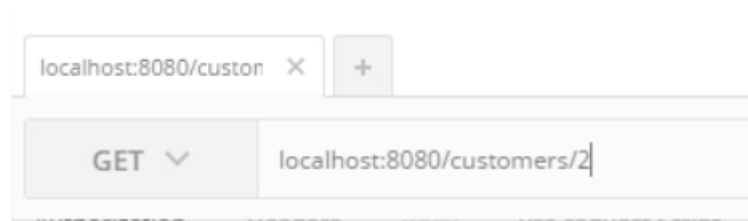
\_\_6. If you still have 'App.java' running from the previous test, stop it using the red stop button in the console window.



\_\_7. Run 'Maven install'.

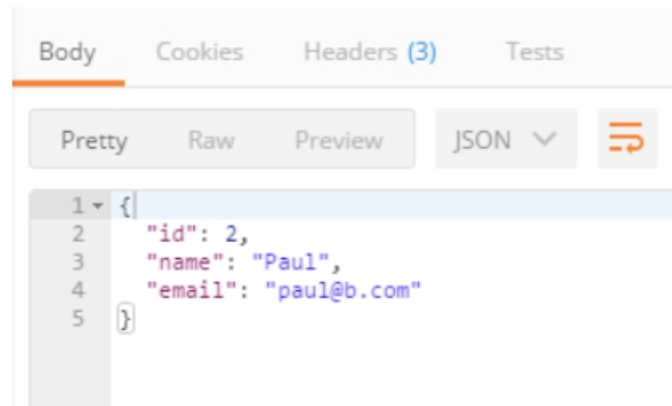
\_\_8. Run 'App.java'.

\_\_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;
}
```

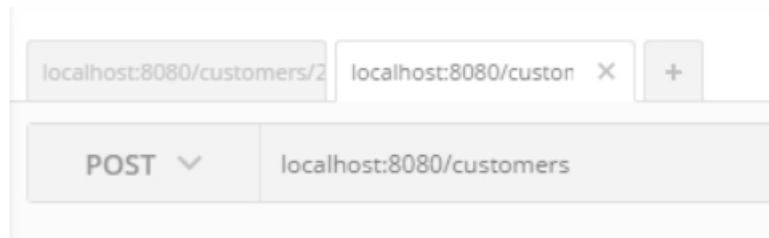
\_\_2. Organize imports.

\_\_3. Save the file.

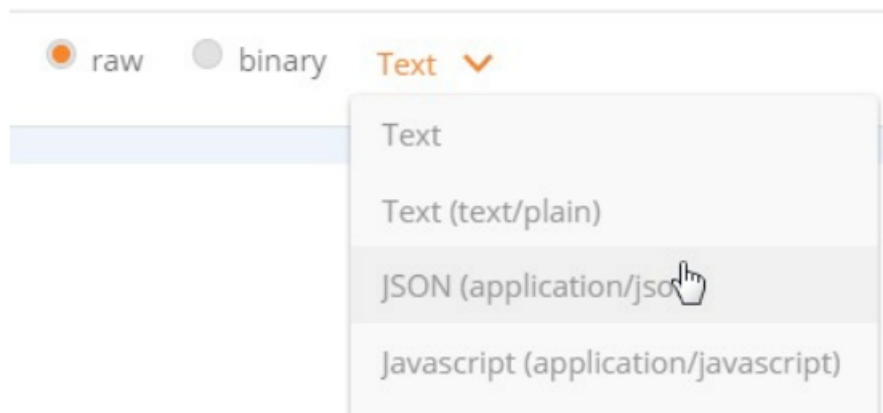
\_\_4. Run 'Maven install'.

\_\_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)'

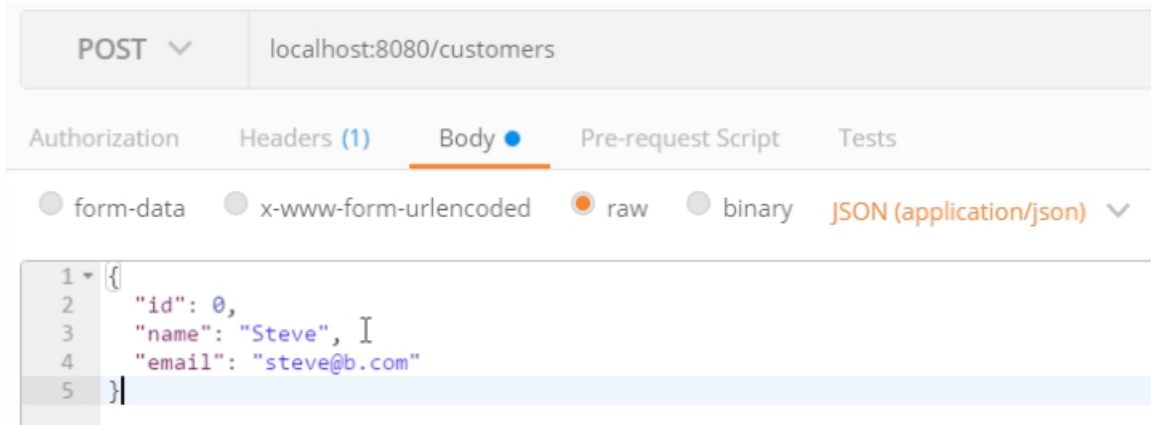


- \_\_11. Enter the following new customer object in the body text area (hint: you might find it convenient to copy and edit an object from the response to the 'GET' request that we issued earlier - it's in the other tab in **Postman** ).

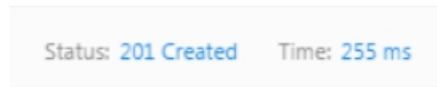
```
1 {  
2   "id": 0,  
3   "name": "Steve",  
4   "email": "steve@b.com"  
5 }
```

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

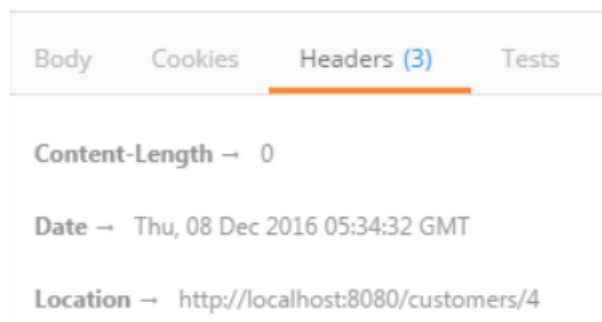
\_\_12. When it looks like below click **Send** .



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



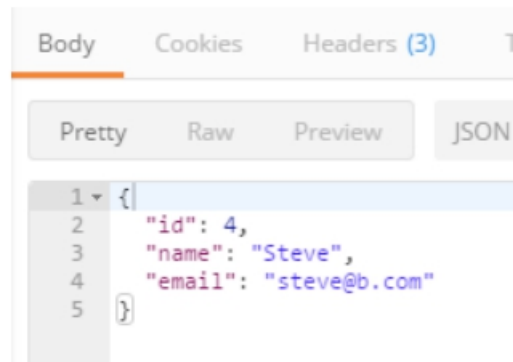
\_\_13. Click on the **Headers** tab in the response area [bottom section]. You should see the 'Location' header:



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.

```
@PutMapping("/{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'.

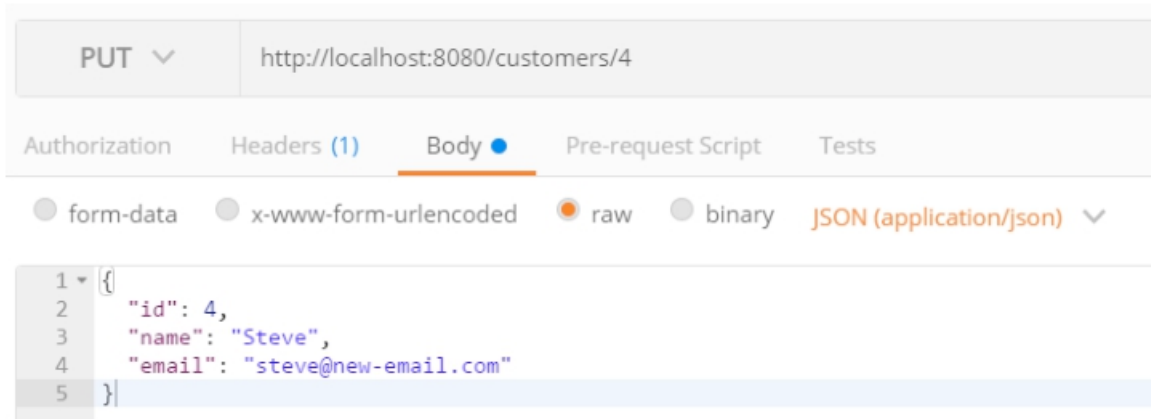


\_\_10. Expand the drop-down box and select 'JSON(application/json)'

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

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

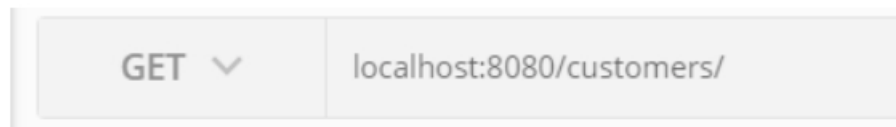
\_\_12. When looks like below click **Send** .



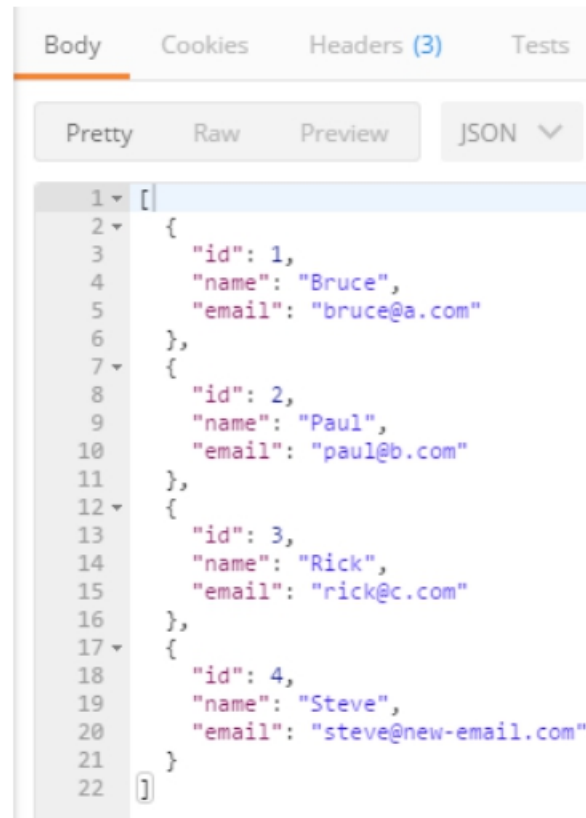
You should see a '200 OK' response.



\_\_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.

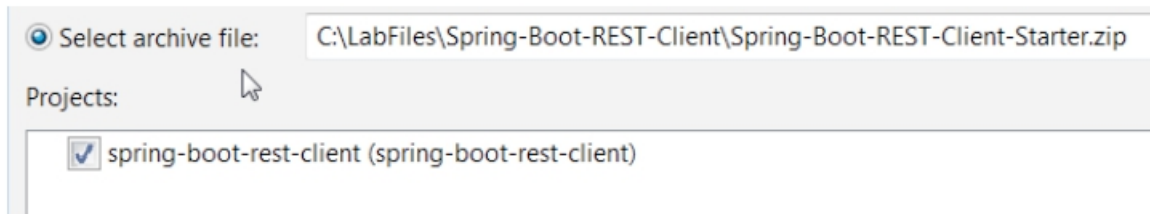
## Lab 6 - Create a RESTful Client with Spring Boot

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.kiddcorp.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:

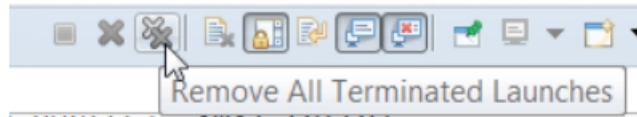
## Browse Purchases

[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**

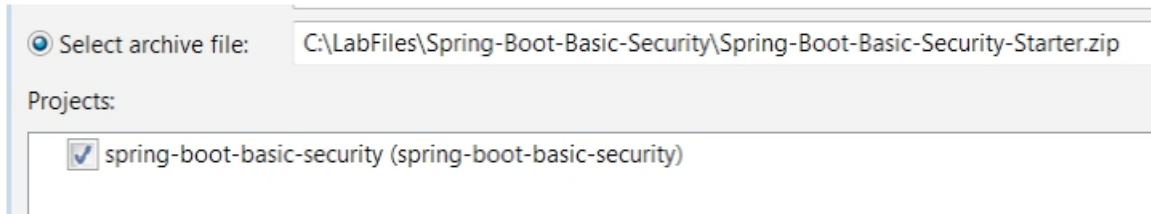
In this lab, we used Spring's RestTemplate object to quickly implement a client to a RESTful service.

## Lab 7 - 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** .

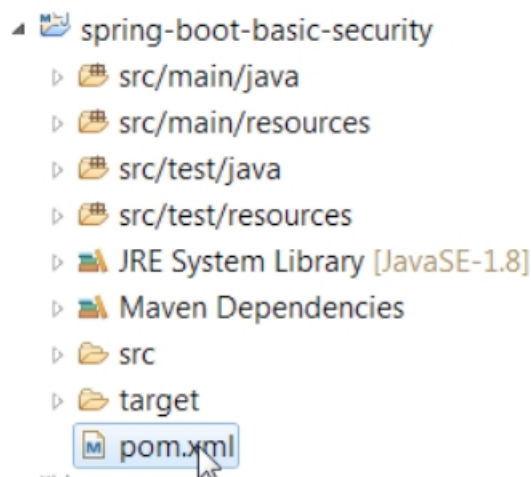


- \_\_\_4. On the **Import** dialog, leave the defaults as-is and click **Finish** .

### 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>
```

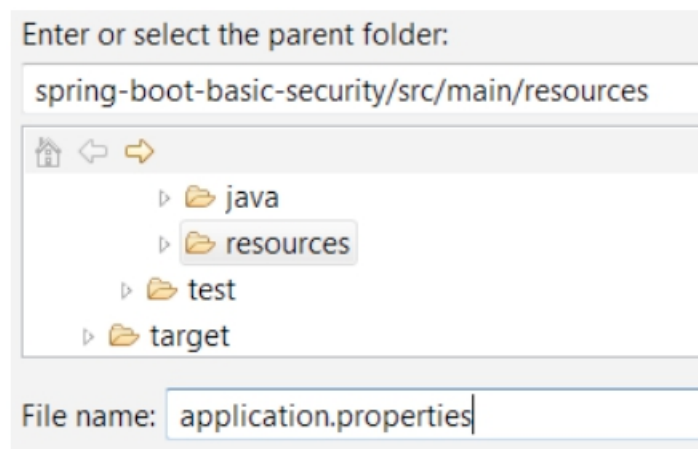
\_\_3. Save and close the file.

### 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**.

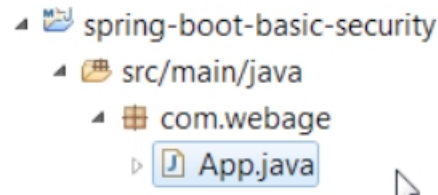
\_\_2. Make sure **spring-boot-basic-security** is selected and click OK.

\_\_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



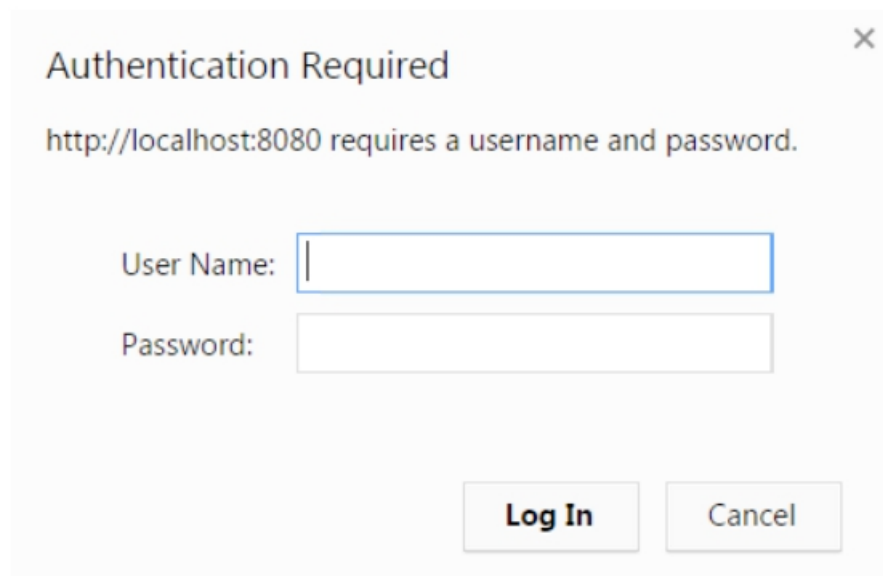
the Maven Install two times to get the Build Success message.

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



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

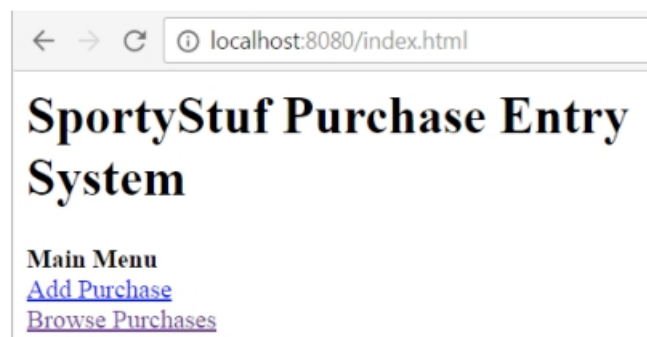
\_\_6. You will be prompted for a user name and password.



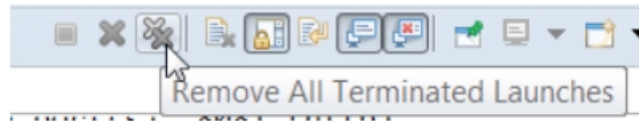
\_\_7. Enter 'user' as the username and 'Pa\$\$w0rd' as the password and then click Login.

\_\_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

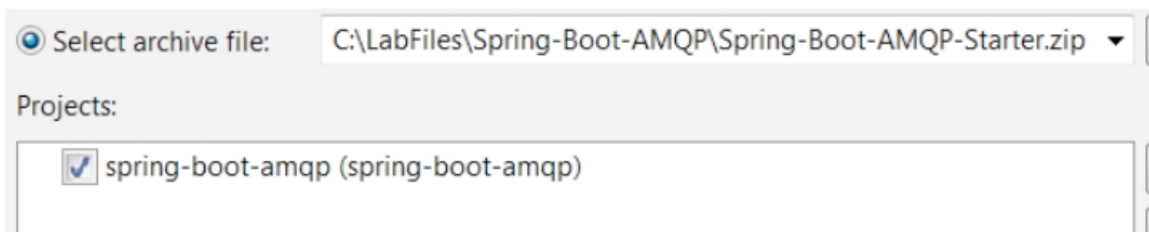
This was a very short lab demonstrating how simple it is to add basic security to an application that is built using Spring Boot.

## Lab 8 - 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:

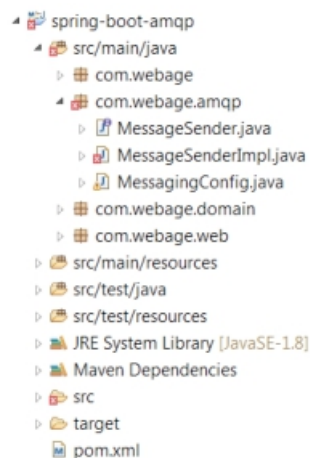
```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

- 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.kiddcorp.amqp' package contains starter point files for the messaging implementation.
- The 'com.kiddcorp.web' package contains a request handler class that receives the form submittal and calls an implementation of 'com.kiddcorp.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.kiddcorp.amqp' package node.



\_\_2. Inside the 'com.kiddcorp.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.kiddcorp.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.kiddcorp.amqp' package.
- \_\_2. Edit the class so the body of it looks like:

```
@Component
public class MyListener {

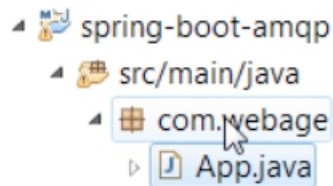
    @RabbitListener(queues="myqueue1")
    public void onMessage(Order order) {
        System.out.println("Message received - Order Details:" + order);
    }
}
```

- \_\_3. Organize imports by pressing **Ctrl-Shift-O** . Select **com.kiddcorp.domain.Order**.
- \_\_4. Save the file.

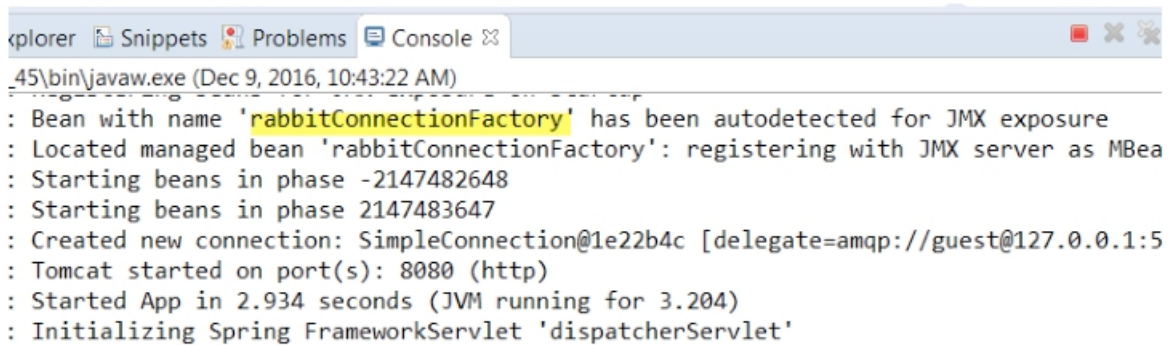
The '@RabbitListener' annotation will establish this method as a listener on the designated queue. Spring Boot will setup an additional thread to monitor the queue and then call this method when a message arrives. Here, the message handler method simply prints out the received message to the console.

## Part 6 - Test

- \_\_1. Run 'Maven Install'.
- \_\_2. Run the 'App.java' class as a Java application.

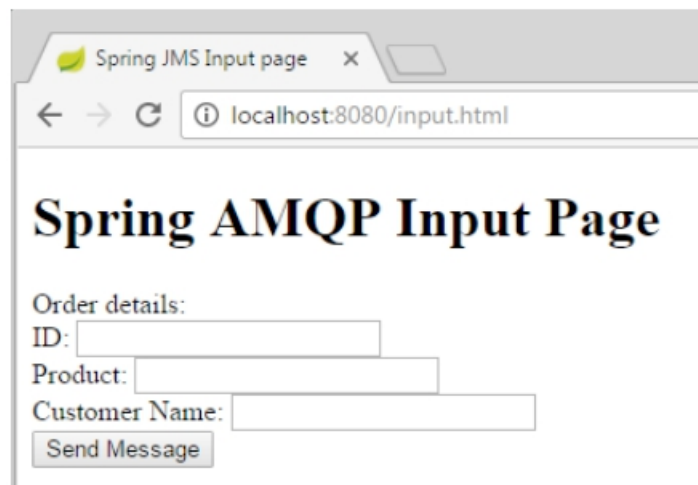


\_\_3. Notice that rabbit has been called.



```
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'.

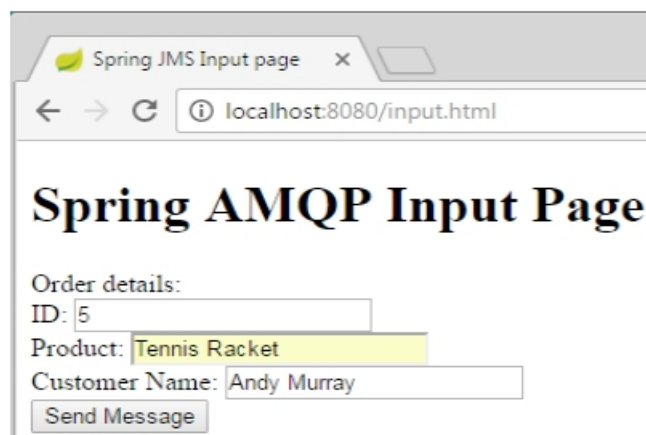


\_\_5. Enter in the following order details:

ID: 5

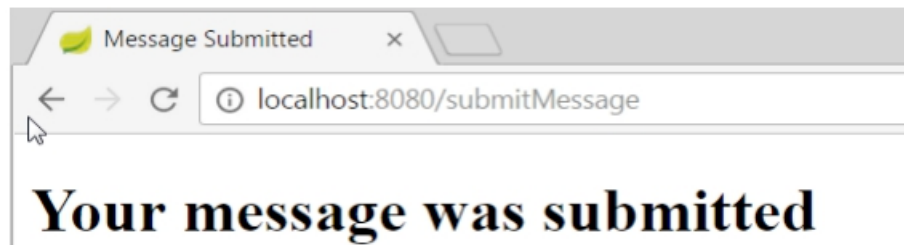
Product: Tennis Racket

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.