Spring Boot Overview:

Spring Boot is an amazing Spring tool created by Pivotal and released in April 2014 (GA). It was developed based on request of SPR-9888 (https://jira.spring.io/browse/SPR-9888) with the title "***Improved support for 'containerless' web application architectures"***

Spring Boot is a utility framework from the Spring team to bootstrap Spring-based applications and micro services quickly and easily. The framework uses **an opinionated approach** over configurations for decision making, thereby reducing the effort required in writing a lot of boilerplate code and configurations.

Building the applications with the spring framework used to be a tedious task, especially when starting a new project.

The Problems with the Traditional Spring Applications are:

1) spring consists of multiple modules like Spring Web MVC, Spring DAO, Spring ORM etc. Which we can use individually or integrate one into another. This integration configurations we always had to do either in XML approach or in annotation based approach.

2) When configuring these modules dependencies in pom.xml file, we need to ensure the jar version compatibility.

3) While integrating the above modules we need to ensure the all module specific custom components (java classes) are available for component-scan so that they can eligible for auto wiring.

4) The application developed with the above modules must need to deploy in another external container like Apache tomcat.

The above problems were addressed with the Spring Boot

Features Of Spring Boot

The above stated problems were eliminated by the spring boot with the following features:

1) **Auto Configuration:**

As a solution for the problem stated in the problem-(1), Spring boot anatomically configures the modules, meaning we no need to configure data source, transaction manager, dispatcher server , handler mappings etc. Just we need to enable either **@EnableAutoConfiguration** or **@SpringBootApplication.**

2) **Spring Boot Starters:**

Spring Boot provides various "Starters" for integrating and auto-configuring other modules of the spring framework. We just need to add the starters as a dependency in the pom file and Spring Boot will automatically configure it using default values.

3) **Spring Boot Actuators:**

Spring Boot actuators provide an excellent out-of-the-box mechanism to monitor and manage Spring Boot applications in production. The Actuator Provides the details like which beans have been configured, bean dependencies, auto config etc details.

Some of the important links Provided by the Spring Boot Actuator are listed as follows:

autoconfig: This displays the autoconfiguration report

mappings: This lists all the HTTP request mappings

info: This displays information about the application

health: This displays the application's health conditions

metrics: This shows different metrics collected from the application

dump: This performs a thread dump and displays the result

4) **Embedded Servlet Container:**

By default, Spring Boot supports Tomcat,Jetty, and Undertow as an embedded Servlet containers. Hence we no need to deploy the application on external servers.

Developing a Restful service – the legacy approach

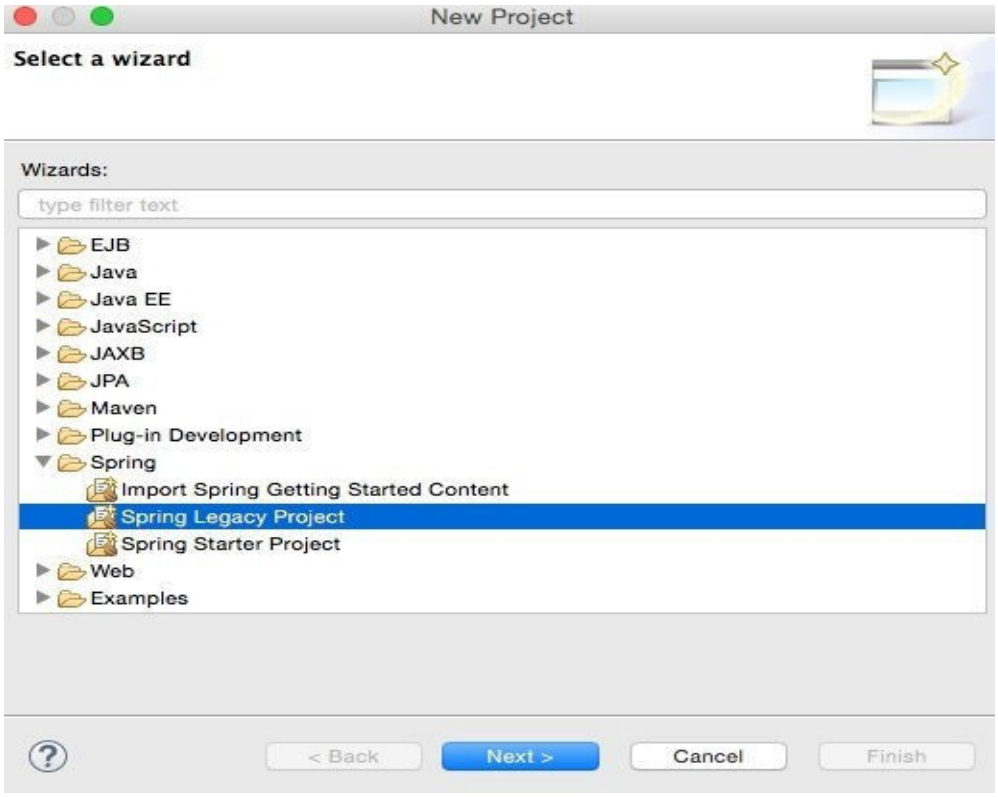
This example will review the traditional RESTful service development before jumping deep into Spring Boot. STS will be used to develop this REST/JSON service.

The following are the steps to develop the first RESTful service:

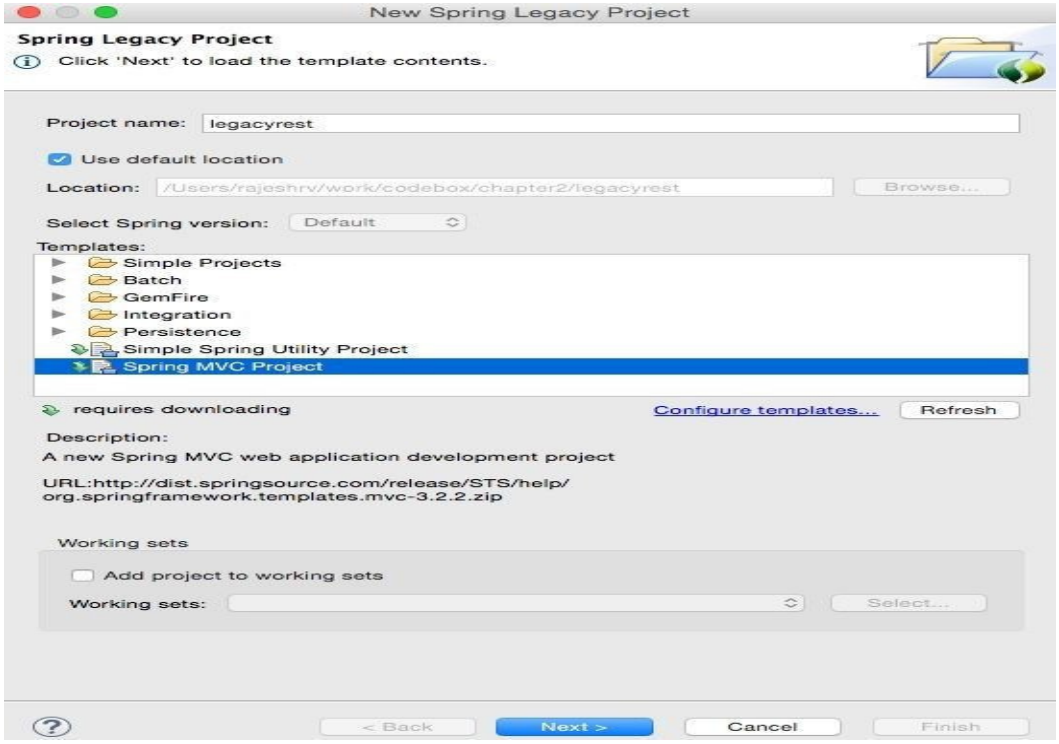
1. Start STS and set a workspace of choice for this project.

2. Navigate to **File** | **New** | **Project**.

3. Select **Spring Legacy Project** as shown in the following screenshot and click on **Next**:



4. Select **Spring MVC Project** as shown in the following diagram and click on **Next**:



5 .Select a top-level package name of choice.

6. Then, click on **Finish**.

7. This will create a project in the STS workspace with the name legacyrest.

Before proceeding further, pom.xml needs editing.

8. Change the Spring version to 4.2.6.RELEASE, as follows:

<org.springframework-version>4.2.6.RELEASE</org.springframework-version>

9. Add **Jackson** dependencies in the pom.xml file for JSON-to-POJO and POJO-to-JSON conversions.

Note that the 2.*.* version is used to ensure compatibility with Spring 4.

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.6.4</version>

</dependency>

10. Some Java code needs to be added. In **Java Resources**, under **legacyrest**, expand the package and open the default **HomeController.java** file:

11. The default implementation is targeted more towards the MVC project. Rewriting HomeController.java to return a JSON value in response to the REST call will do the trick. The resulting HomeController.java file will look similar to the following:

@RestController

```
public class HomeController {

@RequestMapping("/")

public Greet sayHello(){

return new Greet("Hello World!");

}

}

class Greet {

private String message;

public Greet(String message) {

this.message = message;

}

//add getter and setter

}
```

Examining the code, there are now two classes:

Greet: This is a simple Java class with getters and setters to represent a data object. There is

only one attribute in the Greet class, which is message.

HomeController.java: This is nothing but a Spring controller REST endpoint to handle HTTP

requests. Note that the annotation used in HomeController is @RestController, which automatically injects @Controller and @ResponseBody and has the same effect as the following code:

```
@Controller

@ResponseBody

public class HomeController { }
```

12. The project can now be run by right-clicking on **legacyrest**, navigating to **Run As | Run On Server**,and then selecting the default server (**Pivotal tc Server Developer Edition v3.1**) that comes along with STS.

This should automatically start the server and deploy the web application on the TC server.

If the server started properly, the following message will appear in the console:

INFO : org.springframework.web.servlet.DispatcherServlet - FrameworkServlet

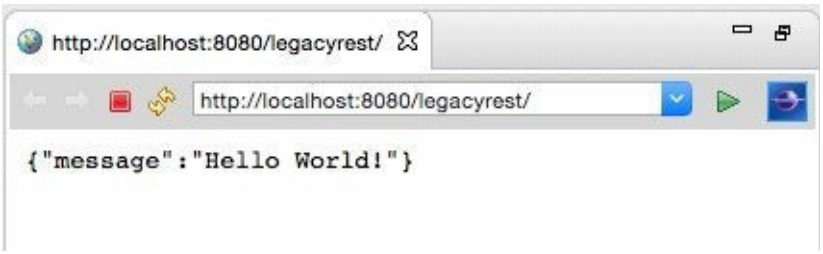
'appServlet': initialization completed in 906 ms

May 08, 2016 8:22:48 PM org.apache.catalina.startup.Catalina start

INFO: Server startup in 2289 ms

13. If everything is fine, STS will open a browser window to **http://localhost:8080/legacyrest/**

and display the JSON object as shown in the browser. Right-click on and navigate to **legacyrest** | **Properties** | **Web Project Settings** and review **Context Root** to identify the context root of the web application:



Moving from traditional web applications to micro services:

The preceding RESTful service is a fully qualified inter-operable REST/JSON service.

However, it is not fully **autonomous** in nature. This is primarily because the service relies on an underlying application server or web container. In the preceding example, a war was explicitly created and deployed on a Tomcat server.

This is a traditional approach to developing RESTful services as a web application. However, from the micro services point of view, one needs a mechanism to develop services as **executable, self-contained JAR files with an embedded HTTP listener**.

Spring Boot is a utility framework from the Spring team to bootstrap Spring-based applications and micro services quickly and easily. The framework uses an **opinionated approach over configurations** for decision making, thereby reducing the effort required in writing a lot of boilerplate code and configurations.

Spring Boot not only increases the speed of development but also provides a set of production-ready ops features such as health checks and metrics collection.Spring Boot recognizes the nature of the application based on the libraries available in the class path and runs the auto configuration classes packaged in these libraries.

There are different ways that Spring Boot-based application development can be started:

- ✓ Using the Spring Boot CLI as a command-line tool

- ✓ Using IDEs such as STS to provide Spring Boot, which are supported out of the box
- ✓ Using the Spring Initialize project at **http://start.spring.io**

Creating a new Spring Boot Project using the CLI

Using the Spring Boot CLI, a command-line tool. Perform the following steps:

1. Install the Spring Boot command-line tool by downloading the spring-boot-cli-1.3.5.RELEASEbin.zip file from http://repo.spring.io/release/org/springframework/boot/spring-bootcli/1.3.5.RELEASE/spring-boot-cli-1.3.5.RELEASE-bin.zip.

2. Unzip the file into a directory of your choice. Open a terminal window and change the terminal prompt to the bin folder. Ensure that the bin folder is added to the system path so that Spring Boot can be run from any location.

3. Verify the installation with the following command. If successful, the Spring CLI version will be printed in the console:

$spring –-version

Spring CLI v1.3.5.RELEASE

4. As the next step, a quick REST service will be developed, which is supported out of the box in Spring Boot. To do so, copy and paste the following code using any editor of choice and save

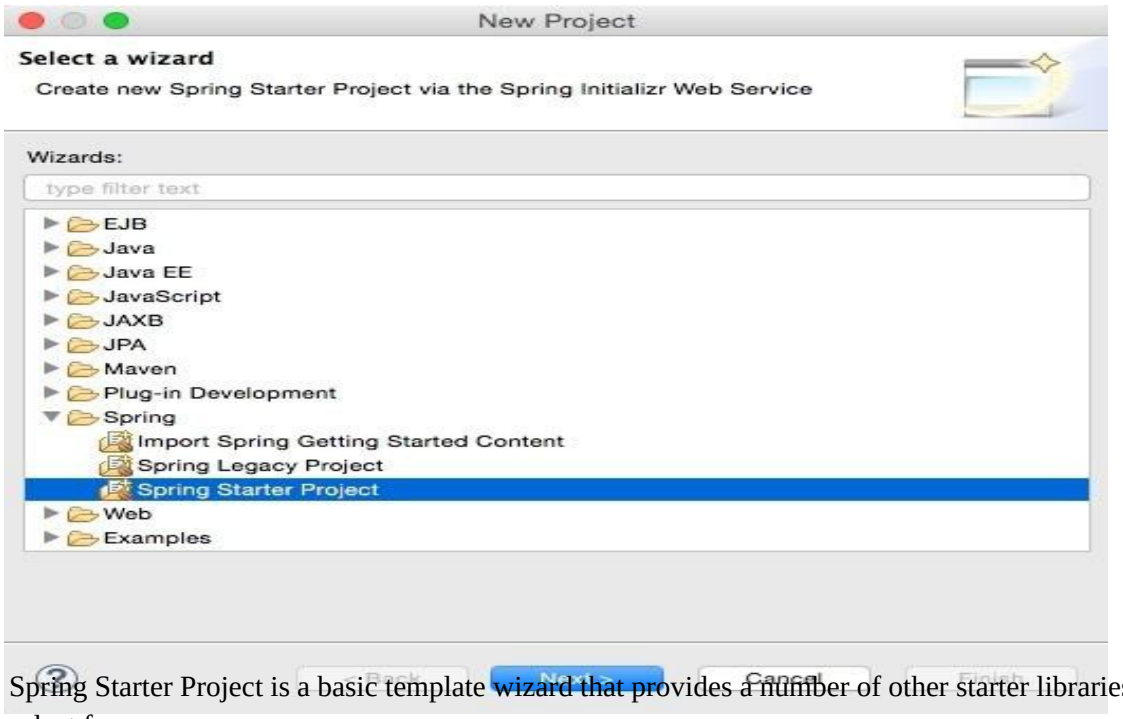
```
@RestController
public class LoginController{

    @RequestMapping(value="/login")
    public String getLogin(){

        return "this is login page";
    }


}
```

In order to run this Java application, go to the folder where LoginController.java is saved and execute the following command. The last few lines of the server start-up log will be similar to the following:

$spring run LoginController.java

Creating a new Spring Boot Project using the STS

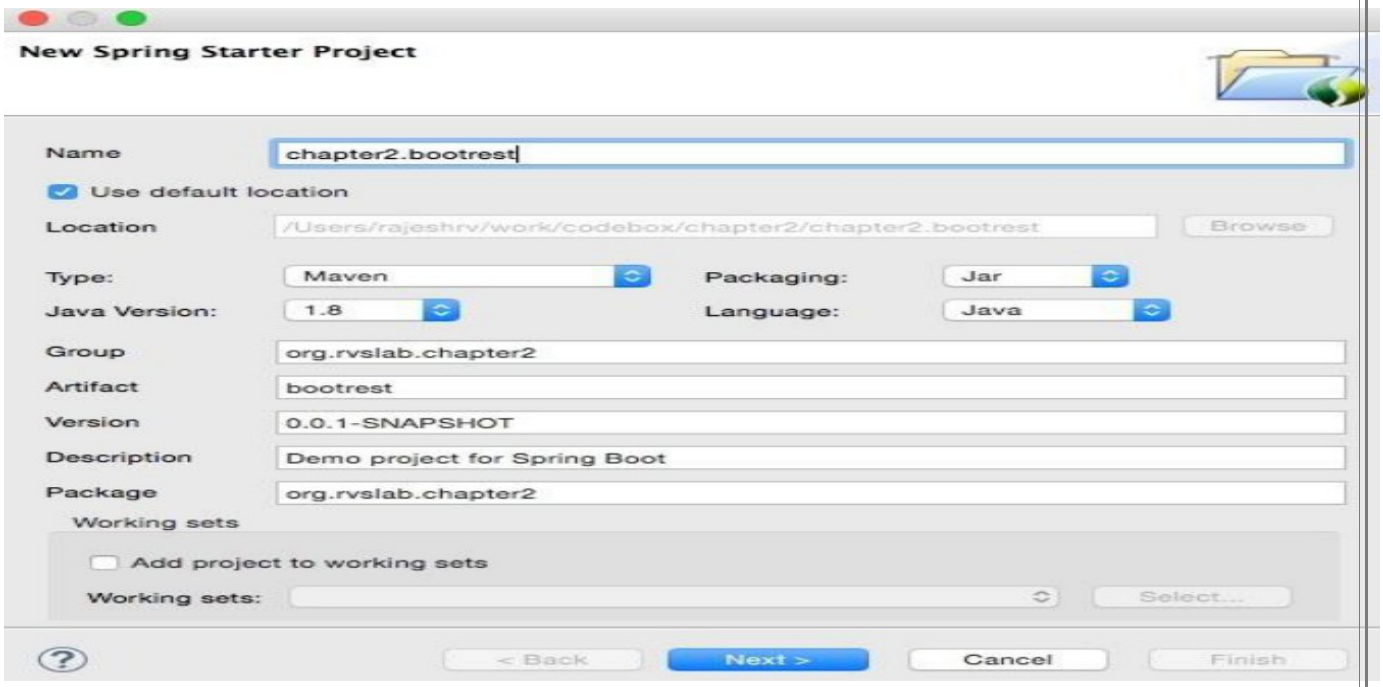
1. Open STS, right-click within the **Project Explorer** window, navigate to **New** | **Project**, and select **Spring Starter Project**, as shown in the following screenshot, and click on **Next**:



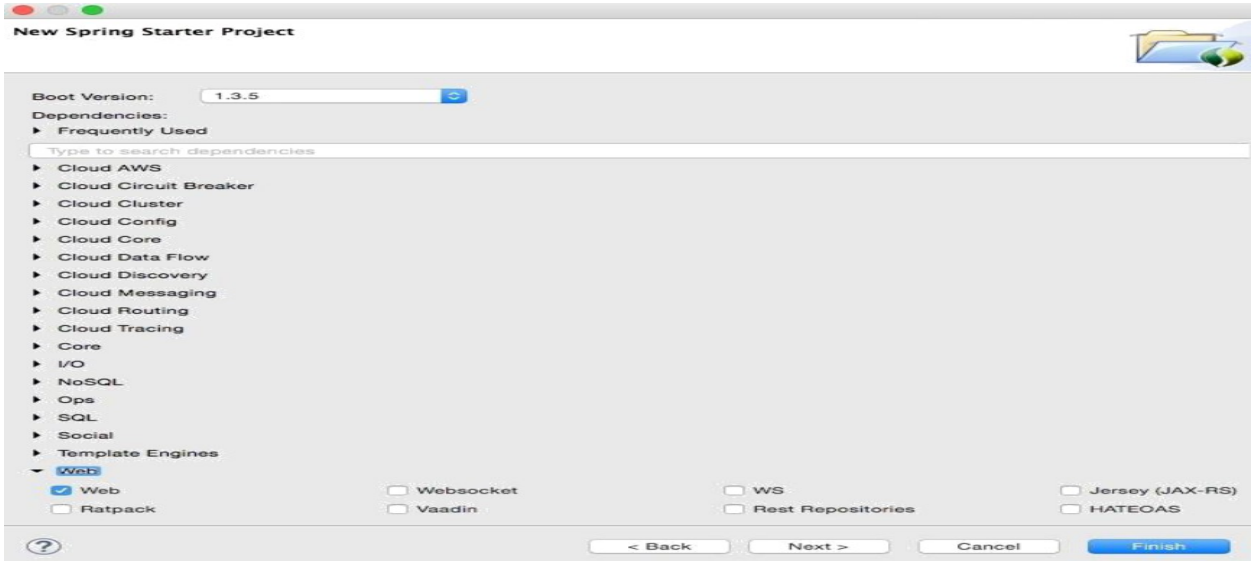
Spring Starter Project is a basic template wizard that provides a number of other starter libraries to select from.

2.Type the project name as chapter2.bootrest or any other name of your choice. It is important to choose the packaging as JAR. In traditional web applications, a war file is created and then deployed to a servlet container, whereas Spring Boot packages all the dependencies to a self-contained, autonomous JAR file with an embedded HTTP listener.

3.Select 1.8 under Java Version. Java 1.8 is recommended for Spring 4 applications. Change the other Maven properties such as Group, Artifact, and Package, as shown in the following screenshot:



4. Once completed, click on **Next**.
5. The wizard will show the library options. In this case, as the REST service is developed, select **Web** under **Web**. This is an interesting step that tells Spring Boot that a Spring MVC web application is being developed so that Spring Boot can include the necessary libraries, including Tomcat as the HTTP listener and other configurations, as required:



6. Click on **Finish**.

This will generate a project named chapter2.bootrest in **Project Explorer** in STS:

7. Take a moment to examine the generated application. Files that are of interest are:

- ✓ pom.xml
- ✓ Application.java
- ✓ Application.properties
- ✓ ApplicationTests.java

Examining the POM file

The parent element is one of the interesting aspects in the pom.xml file. Take a look at the following:

```
<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>1.3.4.RELEASE</version>
</parent>
```

The **spring-boot-starter-parent** pattern is a bill of materials (BOM), a pattern used by Maven's dependency management. BOM is a special kind of POM file used to manage different library versions required for a project. The advantage of using the spring-boot-starter-parent POM file is that developers need not worry about finding the right compatible versions of different libraries such as Spring, Jersey, JUnit, Logback, Hibernate, Jackson, and so on. For instance, in our first legacy example, a specific version of the Jackson library was added to work with Spring 4. In this example, these are taken care of by the spring-boot-starter-parent pattern.

The starter POM file has a list of Boot dependencies, sensible resource filtering, and sensible plug-in configurations required for the Maven builds. The starter POM file itself does not add JAR dependencies to the project. Instead, it will only add library versions. Subsequently, when dependencies are added to the POM file, they refer to the library versions from this POM file. A snapshot of some of the properties are as shown as follows:

```
<spring-boot.version>1.3.5.BUILD-SNAPSHOT</spring-boot.version>
```

```
<hibernate.version>4.3.11.Final</hibernate.version>
<jackson.version>2.6.6</jackson.version>
<jersey.version>2.22.2</jersey.version>
<logback.version>1.1.7</logback.version>
<spring.version>4.2.6.RELEASE</spring.version>
<spring-data-releasetrain.version>Gosling-SR4</spring-data-releasetrain.version>
<tomcat.version>8.0.33</tomcat.version>
```

Open the pom.xml file created in the above project. this is a clean and neat POM file with only two dependencies, as follows:

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

As web is selected, spring-boot-starter-web adds all dependencies required for a Spring MVC project. It also includes dependencies to Tomcat as an embedded HTTP listener. This provides an effective way to get all the dependencies required as a single bundle. Individual dependencies could be replaced with other libraries, for example replacing Tomcat with Jetty.

Similar to web, Spring Boot comes up with a number of spring-boot-starter-* libraries, such as amqp, aop, batch, data-jpa, thymeleaf, and so on.

The last thing to be reviewed in the pom.xml file is the Java 8 property. By default, the parent POM file adds Java 6. It is recommended to override the Java version to 8 for Spring:

<java.version>1.8</java.version>

Note:

Refer http://repo1.maven.org/maven2/ for complete list of starters and their POM file.

Examine Application.java File

Spring Boot, by default, generated a **Application.java** class under src/main/java to bootstrap, as follows:

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

There is only a main method in Application, which will be invoked at startup as per the Java convention. The main method bootstraps the Spring Boot application by calling the run method on SpringApplication. Application.class is passed as a parameter to tell Spring Boot that this is the primary component.

More importantly, the magic is done by the **@SpringBootApplication** annotation. The @SpringBootApplication annotation is a top-level annotation that encapsulates three other annotations, as shown in the following code snippet:

```
@Configuration
@EnableAutoConfiguration
@ComponentScan
public class Application {}
```

@Configuration: The @Configuration annotation hints that the contained class declares one or more @Bean definitions. The @Configuration annotation is meta-annotated with @Component; therefore, it is a candidate for component scanning.
@EnableAutoConfiguration: The @EnableAutoConfiguration annotation tells Spring Boot to automatically configure the Spring application based on the dependencies available in the class path. For example, If **HSQLDB** is on classpath, and have not manually configured any database connection beans, then Spring will auto-configure an in-memory database.

Examining application.properties

A default application.properties file is placed under **src/main/resources**. It is an important file to configure any required properties for the Spring Boot application.

Examining ApplicationTests.java

The last file to be examined is ApplicationTests.java under **src/test/java**. This is a placeholder to write test cases against the Spring Boot application.

Adding REST End Point

To implement the first RESTful service, add a REST endpoint, as follows:

1. One can edit Application.java under src/main/java and add a RESTful service implementation. The RESTful service is exactly the same as what was done in the previous project.

Append the following code at the end of the Application.java file:

```
@RestController

class GreetingController{

@RequestMapping("/")

Greet greet(){

return new Greet("Hello World!");

}
```

}

class Greet {

private String message;

public Greet() {}

public Greet(String message) {

this.message = message;

}

//add getter and setter

}

2. To run, navigate to **Run As** | **Spring Boot App**. Tomcat will be started on the 8080 port:

In the above code, **@RestController** annotation will mark our class as a Spring MVC Controller providing a restful endpoint. Spring Boot recognizes it and will setup spring MVC and a default JSON transformation using the Jackson library.

From the above image, We can notice from the log that:

- Spring Boot get its own process ID (in this case, it is 41130)
- Spring Boot is automatically started with the Tomcat server at the localhost, port 8080.

3. Next, open a browser and point to http://localhost:8080. This will show the JSON response as shown in the following screenshot:

A key difference between the legacy service and this one is that the **Spring Boot service is self-contained.** To make this clearer, run the Spring Boot application outside STS. Open a terminal window, go to the project folder, and run Maven, as follows:

$ maven install

This will generate a fat JAR file under the target folder of the project. Running the application from the command line shows:

$java -jar target/bootrest-0.0.1-SNAPSHOT.jar

As one can see, **bootrest-0.0.1-SNAPSHOT.jar** is self-contained and could be run as a standalone application. **At this point, the JAR is as thin as 13 MB.** Even though the application is no more than just "Hello World", the Spring Boot service just developed, practically follows the principles of microservices.

Testing the Spring Boot Application

There are multiple ways to test REST/JSON Spring Boot microservices. The easiest way is to use a web browser or a curl command pointing to the URL, as follows:

curl http://localhost:8080

There are number of tools available to test RESTful services, such as Postman, Advanced REST client, SOAP UI, Paw, and so on.

In this example, to test the service, the default test class generated by Spring Boot will be used.

Adding a new test case to ApplicatonTests.java results in:

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = Application.class)
@WebIntegrationTest
public class ApplicationTests {
@Test
public void testVanillaService() {
RestTemplate restTemplate = new RestTemplate();
Greet greet = restTemplate.getForObject("http://localhost:8080", Greet.class);
Assert.assertEquals("Hello World!", greet.getMessage());
  }
}
```

Note:

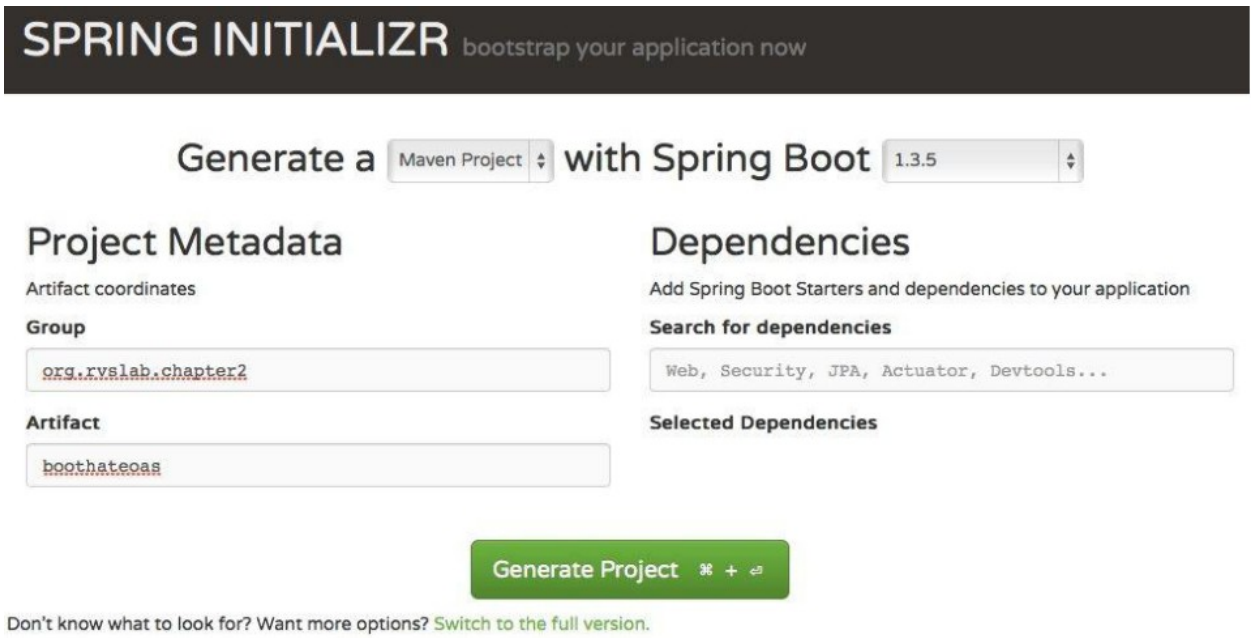
1) **@WebIntegrationTest** is added and @WebAppConfiguration removed at the class level. The @WebIntegrationTest annotation is a handy annotation that ensures
that the tests are fired against a fully up-and-running server. Alternately, a combination of @WebAppConfiguration and @IntegrationTest will give the
same result.

2) RestTemplate is used to call the RESTful service. RestTemplate is a utility class that abstracts the lower-level details of the HTTP client.

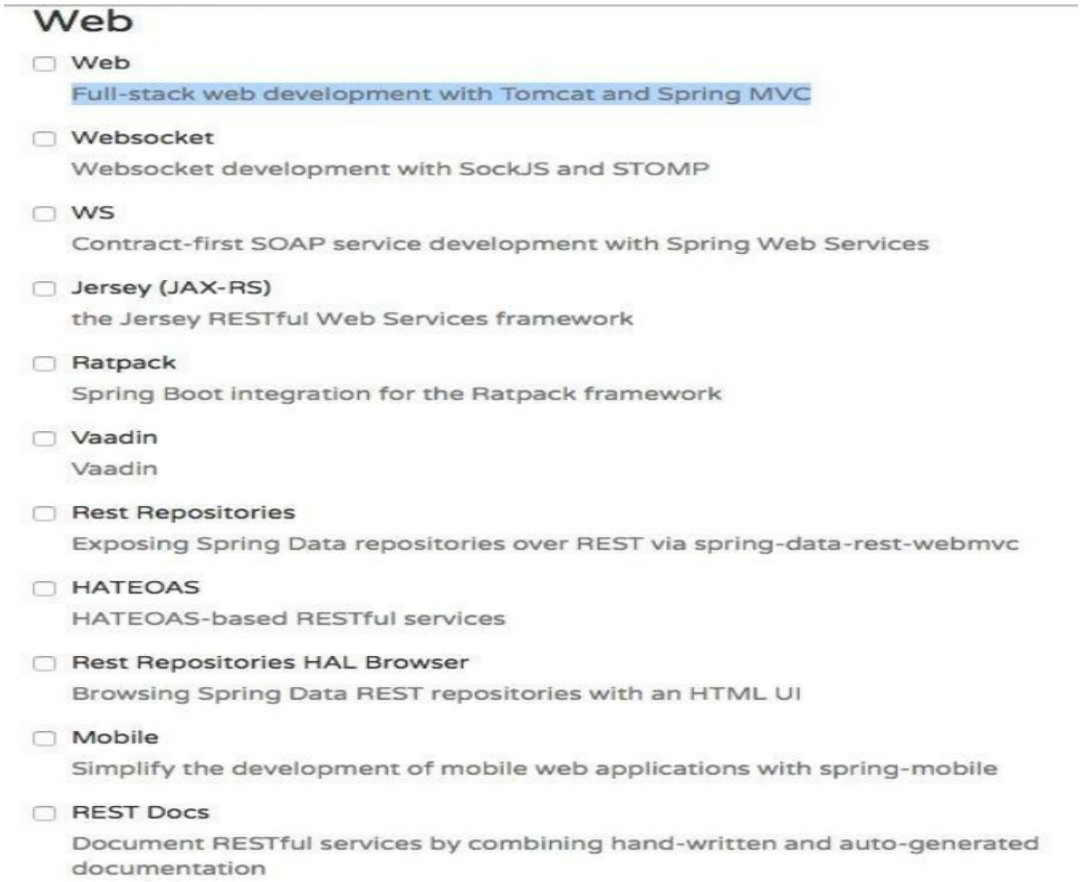
To test this, one can open a terminal window, go to the project folder, and run mvn install.

Creating a new Spring Boot Project using the Spring Initializer:

1. In order to use Spring Initilizr, go to https://start.spring.io:

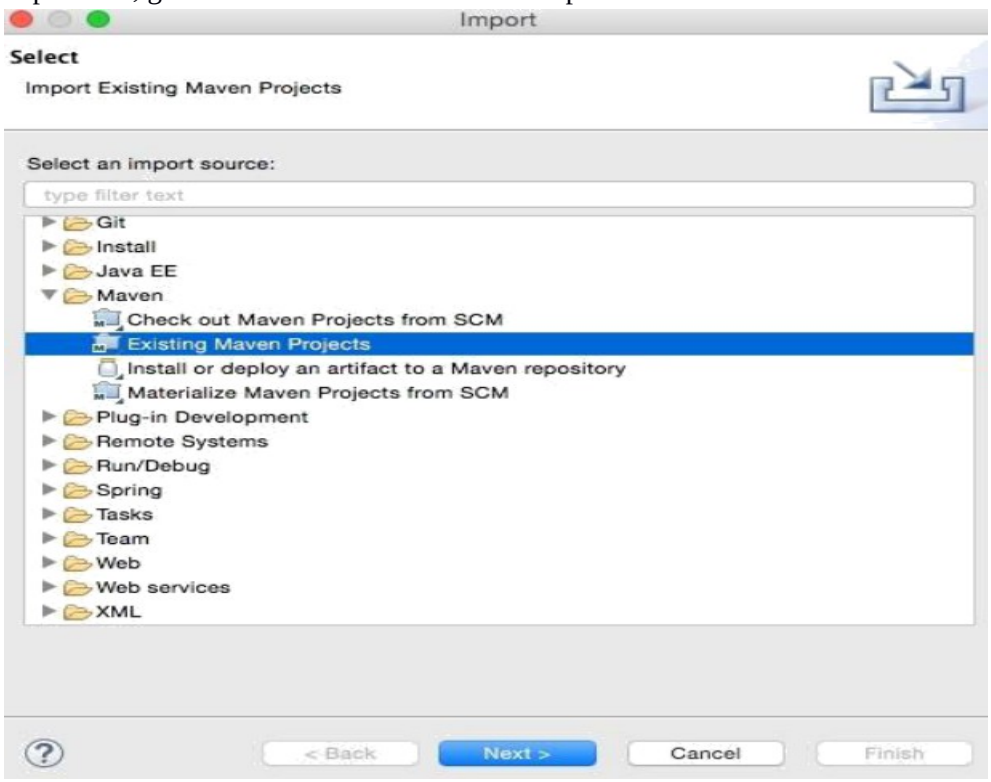


Fill the details, such as whether it is a Maven project, Spring Boot version, group, and artifact ID, as shown earlier, and click on Switch to the full version link under the Generate Project button.Make sure that the Java version is 8 and package type as jar.



Once selected, hit the Generate Project button. This will generate a Maven project and download the project as a ZIP file into the download directory of the browser. Unzip the file and save it to a directory of your choice.

Open STS, go to the File menu and click on Import:

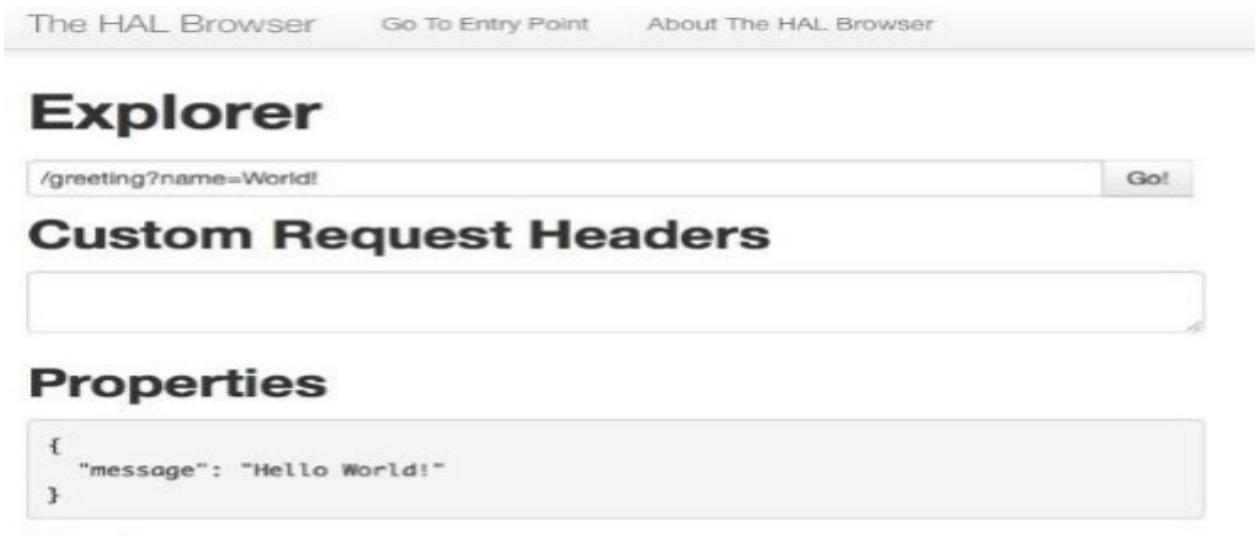


- Navigate to Maven | Existing Maven Projects and click on Next.
- Click on Browse next to Root Directory and select the unzipped folder. Click on Finish. This will load the generated Maven project into STS' Project Explorer.
- Edit the Application.java file to add a new REST endpoint, as follows:

```
@RequestMapping("/greeting")
@ResponseBody
public HttpEntity<Greet> greeting(@RequestParam(value = "name") String name) {
Greet greet = new Greet("Hello " + name);
return new ResponseEntity<Greet>(greet, HttpStatus.OK);
}
```

Note that this is the same GreetingController class as in the previous example.
However, a method was added this time named greeting.

Run this as a Spring Boot app. Once the server startup is complete, point the browser to http://localhost:8080. This will open the browser window. In the Explorer field, type /greeting?name=World! and click on the Go button. If everything is fine, the browser will show the response details as shown in the following screenshot:



Understanding the Spring Boot autoconfiguration

Spring Boot uses convention over configuration by scanning the dependent libraries available in the class path. For each spring-boot-starter-* dependency in the POM file, Spring Boot executes a default AutoConfiguration class. AutoConfiguration classes use the *AutoConfiguration lexical pattern, where * represents the library. For example, the autoconfiguration of JPA repositories is done through **JpaRepositoriesAutoConfiguration.**

Run the application with --debug to see the autoconfiguration report. The following command shows the auto configuration report

D:\UI>java -jar demo-0.0.1-SNAPSHOT.jar --debug

then we could see the auto configuration report with the positive and negative matches.

```
=========================
AUTO-CONFIGURATION REPORT
=========================
Positive matches:
-----------------
DispatcherServletAutoConfiguration matched:
      - @ConditionalOnClass found required class 'org.springframework.web.servlet.DispatcherServlet'; @ConditionalOnMissingClass did not find unwanted class (OnClassCondition)
                  - @ConditionalOnWebApplication (required) found StandardServletEnvironment (OnWebApplicationCondition)

DispatcherServletAutoConfiguration.DispatcherServletConfiguration matched:
                  - @ConditionalOnClass found required class 'javax.servlet.ServletRegistration'; @ConditionalOnMissingClass did not find unwanted class (OnClassCondition)
                        - Default DispatcherServlet did not find dispatcher servlet beans (DispatcherServletAutoConfiguration.DefaultDispatcherServletCondition)
DispatcherServletAutoConfiguration.DispatcherServletRegistrationConfiguration matched:
                  - @ConditionalOnClass found required class 'javax.servlet.ServletRegistration'; @ConditionalOnMissingClass did not find unwanted class (OnClassCondition)
                        - DispatcherServlet Registration did not find servlet registration bean
```

(DispatcherServletAutoConfiguration.DispatcherServletRegistrationCondition)

DispatcherServletAutoConfiguration.DispatcherServletRegistrationConfiguration#dispatcherServletRegistration matched:

- @ConditionalOnBean (names: dispatcherServlet; types: org.springframework.web.servlet.DispatcherServlet; SearchStrategy: all) found beans 'dispatcherServlet', 'dispatcherServlet' (OnBeanCondition)

EmbeddedServletContainerAutoConfiguration matched:

- @ConditionalOnWebApplication (required) found StandardServletEnvironment (OnWebApplicationCondition)

EmbeddedServletContainerAutoConfiguration.EmbeddedTomcat matched:

- @ConditionalOnClass found required classes 'javax.servlet.Servlet', 'org.apache.catalina.startup.Tomcat'; @ConditionalOnMissingClass did not find unwanted class (OnClassCondition)

- @ConditionalOnMissingBean (types: org.springframework.boot.context.embedded.EmbeddedServletContainerFactory; SearchStrategy: current) did not find any beans (OnBeanCondition)

ErrorMvcAutoConfiguration matched:

- @ConditionalOnClass found required classes 'javax.servlet.Servlet', 'org.springframework.web.servlet.DispatcherServlet'; @ConditionalOnMissingClass did not find unwanted class (OnClassCondition)

- @ConditionalOnWebApplication (required) found StandardServletEnvironment (OnWebApplicationCondition)

ErrorMvcAutoConfiguration#basicErrorController matched:

- @ConditionalOnMissingBean (types: org.springframework.boot.autoconfigure.web.ErrorController; SearchStrategy: current) did not find any beans (OnBeanCondition)

ErrorMvcAutoConfiguration#errorAttributes matched:

- @ConditionalOnMissingBean (types: org.springframework.boot.autoconfigure.web.ErrorAttributes; SearchStrategy: current) did not find any beans (OnBeanCondition)

ErrorMvcAutoConfiguration.DefaultErrorViewResolverConfiguration#conventionErrorViewResolver matched:

- @ConditionalOnBean (types: org.springframework.web.servlet.DispatcherServlet; SearchStrategy: all) found bean 'dispatcherServlet'; @ConditionalOnMissingBean (types: org.springframework.boot.autoconfigure.web.DefaultErrorViewResolver; SearchStrategy: all) did not find any beans (OnBeanCondition)

ErrorMvcAutoConfiguration.WhitelabelErrorViewConfiguration matched:

- @ConditionalOnProperty (server.error.whitelabel.enabled) matched (OnPropertyCondition)

- ErrorTemplate Missing did not find error template view (ErrorMvcAutoConfiguration.ErrorTemplateMissingCondition)

ErrorMvcAutoConfiguration.WhitelabelErrorViewConfiguration#**beanNameViewResolver matched:**

- @ConditionalOnMissingBean (types: org.springframework.web.servlet.view.BeanNameViewResolver; SearchStrategy: all) did not find any beans (OnBeanCondition)

ErrorMvcAutoConfiguration.WhitelabelErrorViewConfiguration#defaultErrorView matched:

- @ConditionalOnMissingBean (names: error; SearchStrategy: all) did not find any beans

(OnBeanCondition)
GenericCacheConfiguration matched:
- Cache org.springframework.boot.autoconfigure.cache.GenericCacheConfiguration automatic cache type (CacheCondition)
HttpEncodingAutoConfiguration matched:
- @ConditionalOnClass found required class 'org.springframework.web.filter.CharacterEncodingFilter'; @ConditionalOnMissingClass did not find unwanted class (OnClassCondition)
- @ConditionalOnWebApplication (required) found StandardServletEnvironment (OnWebApplicationCondition)
- @ConditionalOnProperty (spring.http.encoding.enabled) matched (OnPropertyCondition)
HttpEncodingAutoConfiguration#characterEncodingFilter matched:
- @ConditionalOnMissingBean (types: org.springframework.web.filter.CharacterEncodingFilter; SearchStrategy: all) did not find any beans (OnBeanCondition)

HttpMessageConvertersAutoConfiguration matched:
- @ConditionalOnClass found required class 'org.springframework.http.converter.HttpMessageConverter'; @ConditionalOnMissingClass did not find unwanted class (OnClassCondition)
HttpMessageConvertersAutoConfiguration#messageConverters matched:
- @ConditionalOnMissingBean (types: org.springframework.boot.autoconfigure.web.HttpMessageConverters; SearchStrategy: all) did not find any beans (OnBeanCondition)
HttpMessageConvertersAutoConfiguration.StringHttpMessageConverterConfiguration matched:
- @ConditionalOnClass found required class 'org.springframework.http.converter.StringHttpMessageConverter'; @ConditionalOnMissingClass did not find unwanted class (OnClassCondition)
HttpMessageConvertersAutoConfiguration.StringHttpMessageConverterConfiguration#stringHttpMessageConverter matched:
- @ConditionalOnMissingBean (types: org.springframework.http.converter.StringHttpMessageConverter; SearchStrategy: all) did not find any beans (OnBeanCondition)
JacksonAutoConfiguration matched:
- @ConditionalOnClass found required class 'com.fasterxml.jackson.databind.ObjectMapper'; @ConditionalOnMissingClass did not find unwanted class (OnClassCondition)
JacksonAutoConfiguration.Jackson2ObjectMapperBuilderCustomizerConfiguration matched:
- @ConditionalOnClass found required classes 'com.fasterxml.jackson.databind.ObjectMapper', 'org.springframework.http.converter.json.Jackson2ObjectMapperBuilder'; @ConditionalOnMissingClass did not find unwanted class (OnClassCondition)
JacksonAutoConfiguration.JacksonObjectMapperBuilderConfiguration matched:
- @ConditionalOnClass found required classes 'com.fasterxml.jackson.databind.ObjectMapper', 'org.springframework.http.converter.json.Jackson2ObjectMapperBuilder'; @ConditionalOnMissingClass did not find unwanted class (OnClassCondition)
JacksonAutoConfiguration.JacksonObjectMapperBuilderConfiguration#jacksonObjectMapperBuilder

matched:
- @ConditionalOnMissingBean (types: org.springframework.http.converter.json.Jackson2ObjectMapperBuilder; SearchStrategy: all) did not find any beans (OnBeanCondition)

JacksonAutoConfiguration.JacksonObjectMapperConfiguration matched:
- @ConditionalOnClass found required classes 'com.fasterxml.jackson.databind.ObjectMapper', 'org.springframework.http.converter.json.Jackson2ObjectMapperBuilder'; @ConditionalOnMissingClass did not find unwanted class (OnClassCondition)

JacksonAutoConfiguration.JacksonObjectMapperConfiguration#jacksonObjectMapper matched:
- @ConditionalOnMissingBean (types: com.fasterxml.jackson.databind.ObjectMapper; SearchStrategy: all) did not find any beans (OnBeanCondition)

JacksonHttpMessageConvertersConfiguration.MappingJackson2HttpMessageConverterConfiguration matched:
- @ConditionalOnClass found required class 'com.fasterxml.jackson.databind.ObjectMapper'; @ConditionalOnMissingClass did not find unwanted class (OnClassCondition)
- @ConditionalOnProperty (spring.http.converters.preferred-json-mapper=jackson) matched (OnPropertyCondition)
- @ConditionalOnBean (types: com.fasterxml.jackson.databind.ObjectMapper; SearchStrategy: all) found bean 'jacksonObjectMapper' (OnBeanCondition)

JacksonHttpMessageConvertersConfiguration.MappingJackson2HttpMessageConverterConfiguration#mappingJackson2HttpMessageConverter matched:
- @ConditionalOnMissingBean (types: org.springframework.http.converter.json.MappingJackson2HttpMessageConverter; SearchStrategy: all) did not find any beans (OnBeanCondition)

JmxAutoConfiguration matched:
- @ConditionalOnClass found required class 'org.springframework.jmx.export.MBeanExporter'; @ConditionalOnMissingClass did not find unwanted class (OnClassCondition)
- @ConditionalOnProperty (spring.jmx.enabled=true) matched (OnPropertyCondition)

JmxAutoConfiguration#mbeanExporter matched:
- @ConditionalOnMissingBean (types: org.springframework.jmx.export.MBeanExporter; SearchStrategy: current) did not find any beans (OnBeanCondition)

JmxAutoConfiguration#mbeanServer matched:
- @ConditionalOnMissingBean (types: javax.management.MBeanServer; SearchStrategy: all) did not find any beans (OnBeanCondition)

JmxAutoConfiguration#objectNamingStrategy matched:
- @ConditionalOnMissingBean (types: org.springframework.jmx.export.naming.ObjectNamingStrategy; SearchStrategy: current) did not find any beans (OnBeanCondition)

MultipartAutoConfiguration matched:
- @ConditionalOnClass found required classes 'javax.servlet.Servlet', 'org.springframework.web.multipart.support.StandardServletMultipartResolver', 'javax.servlet.MultipartConfigElement'; @ConditionalOnMissingClass did not find unwanted class (OnClassCondition)
- @ConditionalOnProperty (spring.http.multipart.enabled) matched (OnPropertyCondition)

MultipartAutoConfiguration#multipartConfigElement matched:
- @ConditionalOnMissingBean (types: javax.servlet.MultipartConfigElement; SearchStrategy: all) did not find any beans (OnBeanCondition)
MultipartAutoConfiguration#multipartResolver matched:
- @ConditionalOnMissingBean (types: org.springframework.web.multipart.MultipartResolver; SearchStrategy: all) did not find any beans (OnBeanCondition)
NoOpCacheConfiguration matched:
- Cache org.springframework.boot.autoconfigure.cache.NoOpCacheConfiguration automatic cache type (CacheCondition)
PropertyPlaceholderAutoConfiguration#propertySourcesPlaceholderConfigurer matched:
- @ConditionalOnMissingBean (types: org.springframework.context.support.PropertySourcesPlaceholderConfigurer; SearchStrategy: current) did not find any beans (OnBeanCondition)
RedisCacheConfiguration matched:
- Cache org.springframework.boot.autoconfigure.cache.RedisCacheConfiguration automatic cache type (CacheCondition)
ServerPropertiesAutoConfiguration matched:
- @ConditionalOnWebApplication (required) found StandardServletEnvironment (OnWebApplicationCondition)
ServerPropertiesAutoConfiguration#serverProperties matched:
- @ConditionalOnMissingBean (types: org.springframework.boot.autoconfigure.web.ServerProperties; SearchStrategy: current) did not find any beans (OnBeanCondition)
SimpleCacheConfiguration matched:
- Cache org.springframework.boot.autoconfigure.cache.SimpleCacheConfiguration automatic cache type (CacheCondition)
ValidationAutoConfiguration matched:
- @ConditionalOnClass found required class 'javax.validation.executable.ExecutableValidator'; @ConditionalOnMissingClass did not find unwanted class (OnClassCondition)
- @ConditionalOnResource found location classpath:META-INF/services/javax.validation.spi.ValidationProvider (OnResourceCondition)
ValidationAutoConfiguration#defaultValidator matched:
- @ConditionalOnMissingBean (types: javax.validation.Validator; SearchStrategy: all) did not find any beans (OnBeanCondition)
ValidationAutoConfiguration#methodValidationPostProcessor matched:
- @ConditionalOnMissingBean (types: org.springframework.validation.beanvalidation.MethodValidationPostProcessor; SearchStrategy: all) did not find any beans (OnBeanCondition)

WebClientAutoConfiguration.RestTemplateConfiguration matched:
- @ConditionalOnClass found required class ='org.springframework.web.client.RestTemplate'; @ConditionalOnMissingClass did not find unwanted class (OnClassCondition)
WebClientAutoConfiguration.RestTemplateConfiguration#restTemplateBuilder matched:
- @ConditionalOnMissingBean (types: org.springframework.boot.web.client.RestTemplateBuilder;

SearchStrategy: all) did not find any beans (OnBeanCondition)

WebMvcAutoConfiguration matched:

- @ConditionalOnClass found required classes 'javax.servlet.Servlet', 'org.springframework.web.servlet.DispatcherServlet', 'org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter'; @ConditionalOnMissingClass did not find unwanted class (OnClassCondition)
- @ConditionalOnWebApplication (required) found StandardServletEnvironment (OnWebApplicationCondition)
- @ConditionalOnMissingBean (types: org.springframework.web.servlet.config.annotation.WebMvcConfigurationSupport; SearchStrategy: all) did not find any beans (OnBeanCondition)

WebMvcAutoConfiguration#hiddenHttpMethodFilter matched:

- @ConditionalOnMissingBean (types: org.springframework.web.filter.HiddenHttpMethodFilter; SearchStrategy: all) did not find any beans (OnBeanCondition)

WebMvcAutoConfiguration#httpPutFormContentFilter matched:

- @ConditionalOnProperty (spring.mvc.formcontent.putfilter.enabled) matched (OnPropertyCondition)
- @ConditionalOnMissingBean (types: org.springframework.web.filter.HttpPutFormContentFilter; SearchStrategy: all) did not find any beans (OnBeanCondition)

WebMvcAutoConfiguration.WebMvcAutoConfigurationAdapter#defaultViewResolver matched:

- @ConditionalOnMissingBean (types: org.springframework.web.servlet.view.InternalResourceViewResolver; SearchStrategy: all) did not find any beans (OnBeanCondition)

WebMvcAutoConfiguration.WebMvcAutoConfigurationAdapter#requestContextFilter matched:

- @ConditionalOnMissingBean (types: org.springframework.web.context.request.RequestContextListener,org.springframework.web.filter.RequestContextFilter; SearchStrategy: all) did not find any beans (OnBeanCondition)

WebMvcAutoConfiguration.WebMvcAutoConfigurationAdapter#viewResolver matched:

- @ConditionalOnBean (types: org.springframework.web.servlet.ViewResolver; SearchStrategy: all) found beans 'defaultViewResolver', 'beanNameViewResolver', 'mvcViewResolver'; @ConditionalOnMissingBean (names: viewResolver; types: org.springframework.web.servlet.view.ContentNegotiatingViewResolver; SearchStrategy: all) did not find any beans (OnBeanCondition)

WebMvcAutoConfiguration.WebMvcAutoConfigurationAdapter.FaviconConfiguration matched:

- @ConditionalOnProperty (spring.mvc.favicon.enabled) matched (OnPropertyCondition)

WebSocketAutoConfiguration matched:

- @ConditionalOnClass found required classes 'javax.servlet.Servlet', 'javax.websocket.server.ServerContainer'; @ConditionalOnMissingClass did not find unwanted class (OnClassCondition)
- @ConditionalOnWebApplication (required) found StandardServletEnvironment (OnWebApplicationCondition)

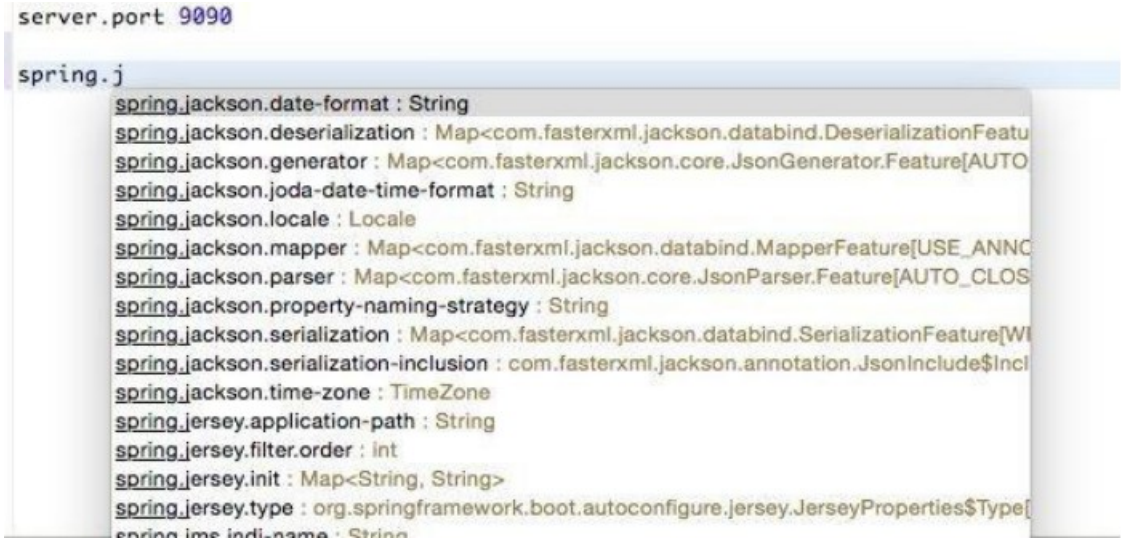
WebSocketAutoConfiguration.TomcatWebSocketConfiguration matched:

- @ConditionalOnClass found required classes 'org.apache.catalina.startup.Tomcat', 'org.apache.tomcat.websocket.server.WsSci'; @ConditionalOnMissingClass did not find unwanted class

(OnClassCondition)
WebSocketAutoConfiguration.TomcatWebSocketConfiguration#websocketContainerCustomizer matched:
- @ConditionalOnJava (1.7 or newer) found 1.8 (OnJavaCondition)
- @ConditionalOnMissingBean (names: websocketContainerCustomizer; SearchStrategy: all) did not find any beans (OnBeanCondition)

It is possible to exclude the autoconfiguration of certain libraries if the application has special requirements and if we wants to get full control of the configurations. The following is an example of excluding DataSourceAutoConfiguration:
@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})

Overriding the default configuration values
It is also possible to override default configuration values using the application.properties file.
STS provides an easy-to-autocomplete, contextual help on application.properties, as shown in the following screenshot:



In the preceding screenshot, server.port is edited to be set as 9090. Running this application again will start the server on port 9090.

Changing the location of the configuration file:
According to the Twelve-Factor app principles,configuration parameters need to be externalized from the code. Spring Boot externalizes all configurations into application.properties. However, it is still part of the application's build. Furthermore, properties can be read from outside the package by setting the following properties:
spring.config.name= # config file name
spring.config.location= # location of config file
Here, spring.config.location could be a local file location.

The following command starts the Spring Boot application with an externally provided configuration file:

$java -jar target/bootadvanced-0.0.1-SNAPSHOT.jar -- spring.config.name=bootrest.properties

Reading the Custom Properties:

At startup, SpringApplication loads all the properties and adds them to the Spring Environment class. Add a custom property to the application.properties file. In this case, the custom property is named **bootrest.customproperty**. Autowire the Spring Environment class into the GreetingController class. Edit the GreetingController class to read the custom property from Environment and add a log statement to print the custom property to the console.
Perform the following steps to do this:

1. Add the following property to the application.properties file:
bootrest.customproperty=hello
2. Then, edit the GreetingController class as follows:

```
@Autowired
Environment env;
Greet greet(){
logger.info("bootrest.customproperty "+
env.getProperty("bootrest.customproperty")); return new Greet("Hello World!");
}
```

3. Rerun the application. The log statement prints the custom variable in the console, as follows:

GreetingController : bootrest.customproperty hello

a) **Using a .yml file for configuration:**

As an alternate to application.properties, one may use a .yaml file. YAML provides a JSON-like structured configuration compared to the flat properties file.To see this in action, simply replace application.properties with application.yaml and add the following property:

```
server:
  port: 8080
```

Rerun the application to see the port printed in the console.

b)Using multiple configuration files(Profiles):

If we want to switch from one environment to another environment without re-build we have to use a seperate properties file for each environment. For instance, lets have dev, stage and prod environments, now we have to maintain three properties files seperately named application-dev.properties, application-stage.properties and application-prod.properties.

But using yml file only one file is enough to define all the environment specific configurations as below:

```yaml
spring:
  profiles: dev

server:
  port: 8080
---
spring:
  profiles: prod
server:
  port: 9090
---
spring:
  profiles:
    active: dev
```

Now clean the application and build using mvn clean install.
And run the generated self contained jar file using profile setting as below:

```
D:\trainings\dotridge\demo\target>java -jar -Dspring.profiles.active=prod demo-0.0.1-SNAPSHOT.jar
```

Now in the console log we could see the profile activation as below:

```
D:\trainings\dotridge\demo\target>java -jar -Dspring.profiles.active=prod demo-0.0.1-SNAPSHOT.jar

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.5.6.RELEASE)

2017-09-10 13:34:59.012  INFO 1308 --- [           main] com.example.demo.DemoApplication         : Starting DemoApplication v0.0.1-SNAPSHOT on IM-RT-
LP-143 with PID 1308 (D:\trainings\dotridge\demo\target\demo-0.0.1-SNAPSHOT.jar started by nsanda in D:\trainings\dotridge\demo\target)
2017-09-10 13:34:59.020  INFO 1308 --- [           main] com.example.demo.DemoApplication         : The following profiles are active: prod
```

If we didn't specify any profile setting like below:

```
D:\trainings\dotridge\demo\target>java -jar demo-0.0.1-SNAPSHOT.jar
```

Then by default profile will be settled to the "dev" environment as shown below:

```
D:\trainings\dotridge\demo\target>java -jar demo-0.0.1-SNAPSHOT.jar

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.5.6.RELEASE)

2017-09-10 13:35:27.242  INFO 7616 --- [           main] com.example.demo.DemoApplication         : Starting DemoApplication v0.0.1-SNAPSHOT on IM-RT-
LP-143 with PID 7616 (D:\trainings\dotridge\demo\target\demo-0.0.1-SNAPSHOT.jar started by nsanda in D:\trainings\dotridge\demo\target)
2017-09-10 13:35:27.249  INFO 7616 --- [           main] com.example.demo.DemoApplication         : The following profiles are active: dev
```

Note:
Active profiles can be specified programmatically using the @ActiveProfiles annotation, which is especially useful when running test cases, as follows:

```java
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.test.context.ActiveProfiles;

@SpringBootApplication
@ActiveProfiles("dev")
public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

Changing the Default Embedded Web Server

Embedded HTTP listeners can easily be customized as follows. By default, Spring Boot supports Tomcat, Jetty, and Undertow. In the following example, Tomcat is replaced with Undertow:

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
      <exclusion>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-tomcat</artifactId>
      </exclusion>
    </exclusions>
</dependency>

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

Enabling Spring Boot Security:

Adding basic authentication to Spring Boot is pretty simple. Add the following dependency to pom.xml. This will include the necessary Spring security library files:

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

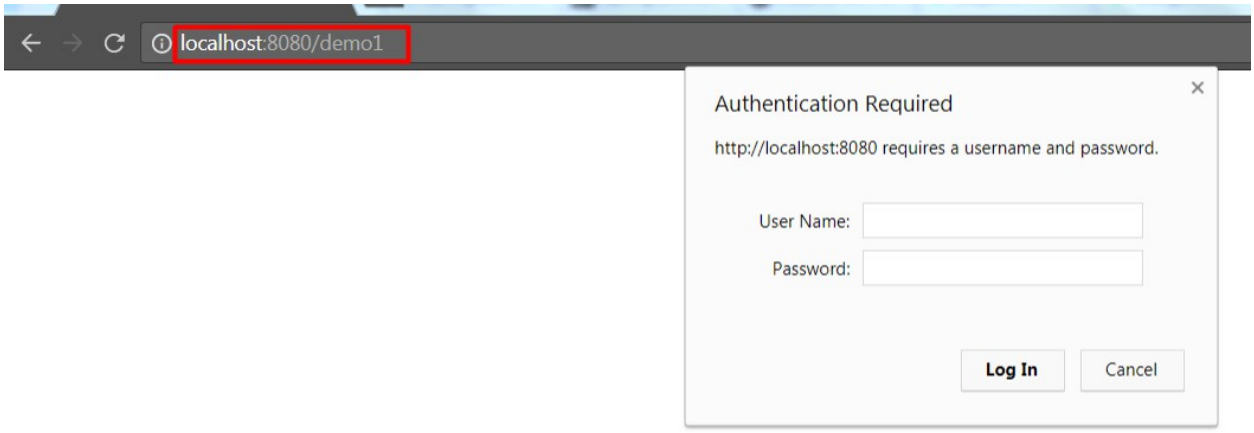
Open Application.java and add @EnableGlobalMethodSecurity to the Application class. This annotation will enable method-level security:

```java
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.security.config.annotation.method.configuration.EnableGlobalMethodSecurity;

@SpringBootApplication
@EnableGlobalMethodSecurity
public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

The default basic authentication assumes the user as being **user**. The default password will be printed in the console at startup , as shown here:

```
type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
2017-09-10 14:06:22.508  INFO 8476 --- [           main] b.a.s.AuthenticationManagerConfiguration :

Using default security password: c34f2018-09db-43b3-8e74-6644381a9800
```

When we run the application, we could see the basic authentication window as below:



Alternately, the user name and password can be added in application.properties, as shown here:

```
security:
  user:
    name: john
    password: testUser@123
```

Add a new test case in ApplicationTests to test the secure service results, as in the following:

```java
package com.example.demo;

import org.junit.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.http.HttpEntity;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpMethod;
import org.springframework.http.ResponseEntity;
import org.springframework.security.crypto.codec.Base64;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.web.client.RestTemplate;

@RunWith(SpringRunner.class)
@SpringBootTest
public class DemoApplicationTests {
    @Test
    public void testSecureService() {
        String plainCreds = "john:testUser@123";
        HttpHeaders headers = new HttpHeaders();
        headers.add("Authorization", "Basic " + new
        String(Base64.encode(plainCreds.getBytes())));
        HttpEntity<String> request = new HttpEntity<String>(headers);
        RestTemplate restTemplate = new RestTemplate();
        ResponseEntity<String> response = restTemplate.exchange("http://localhost:8080/demo1", HttpMethod.GET, request,
                String.class);
        Assert.assertEquals("8080", new String(response.getBody().getBytes()));
    }
}
```

As shown in the code, a new Authorization request header with Base64 encoding the username-password string is created.
Rerun the application using Maven. Note that the new test case passed, but the old test case failed with an exception. The earlier test case now runs without credentials, and as a result, the server rejected the request with the following message:
org.springframework.web.client.HttpClientErrorException: 401 Unauthorized

Data Access Layer With Spring Data JPA

Spring Data is an umbrella project to make working with data stores easier and encapsulating the actual data storage access. Meaning we can switch the database back end without changing a single line of our data access code.
To use in our application we need to add the dependency to the spring boot starter as shown below:

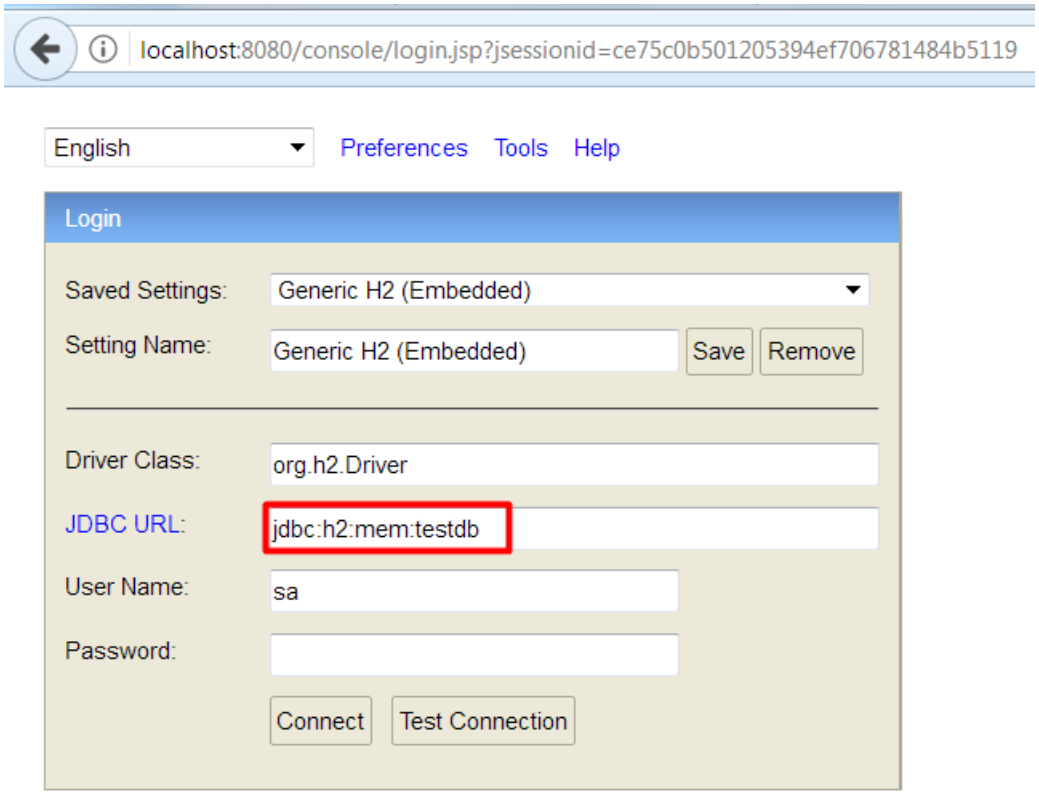
```xml
<!-- Spring Data JPA -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

Now we need to tell spring boot which database we are going to use. The first thing we add is the database JDBC driver as a dependency to the pom. For the sake of simplicity we are going to use the **embedded in-memory database H2.**

```
<!-- JDBC Driver -->
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
</dependency>
```

When we use H2, HSQL or Derby as an embedded database, Spring Boot takes care of setting up whole access layer, whenever we use a different database we must define access to the database. We need to set those configurations in **application.properties/application.yml** file.

When we configured the above h2 database as dependency in the pom.xml file, then as per convention over configuration approach, as @EnableAutoConfiguration enabled, Spring boot automatically establish the database connection as the dependency is available in the class path and creates the tables automatically for the jpa entities when we run the application. The following screen shows the default h2 database details:



Note:

1) By Default the above console will be in disabled state, In order to enabled it we have to configure the below properties in application.properties/.yml file as below:

```
spring.h2.console.enabled=true
spring.h2.console.path=/console
```

Here when we configured the property **"spring.h2.console.path"** , then we could see the above console on http://localhost:8080/console (if our application is running on port:808, and host:localhost)

2) if we wants to see the console log what is happening and how it is writing/reading the quarries wrt. H2 database, we have to configure the property:

```
spring.jpa.show-sql=true
```

As we seen the above console, the default schema of H2 database taken by the spring boot is “testdb”. If we wants to overide this and wants to configure our own schema the the following configurations needs to do in the application.properties / application.yml file as shown below:

```
spring:
  profiles:
    active: dev
  datasource:
    url: jdbc:h2:mem:mydb
    username: sa
```

In application.properties file:

```
spring.datasource.url=jdbc:h2:mem:mydb
spring.datasource.username=sa
```

Here **spring.datasource.url** tells spring boot where the database is located and which driver to use. It follows standard JDBC URL naming scheme. **spring.datasource.username** is the user name to access the database. In the case of an in-memory db, we do not need to add the password, but it would be defined with a property **spring.datasource.password.**

When we start our app now, database access is automatically configured and ready to go. The only thing missing is the database itself, but we can let hibernate create it by add the following to our application.properties file.

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

The property **spring.jpa.hibernate.ddl-auto** defines if and when hibernate will create our database with all tables. In our case, we let it update the schema every time we start the application.

JPA Entity classes are specified in **persistence.xml** file but with spring boot we do not need that tedious task and can use the Entity Scan. By Default all the packages below the configuration class, in our case, **DemoApplication** will be scanned for classes annotated with @Entity, @Embeddable or @MappedSuperclass. If they reside in a different package, we must add **@EntityScan** to our configuration class.

Sample Demo application:
Step-1: Create a JPA Entity class

```java
package com.model;

import java.io.Serializable;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Patient  implements Serializable{

    /**
     *
     */
    private static final long serialVersionUID = 7229385401097928276L;
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int patId;
    private String patName;
    public int getPatId() {
        return patId;
    }
    public void setPatId(int patId) {
        this.patId = patId;
    }
    public String getPatName() {
        return patName;
    }
    public void setPatName(String patName) {
        this.patName = patName;
    }
}
```

Step-2: Create Repository class which extends JPA CrudRespository

```java
package com.example.demo;

import org.springframework.data.repository.CrudRepository;

public interface PatientRepository  extends CrudRepository<Patient, Long>{

}
```

Step-3: As our entity classes in another seperate package i.e. com.model and our DemoApplication.java is in package com.example, we need to add @EntityScan annotation on DemoApplication.java as below:

```
import com.model.Patient;

@SpringBootApplication
@EntityScan("com.model")
public class DemoApplication
```

<u>Step-4</u>: Implements CommandLineRunner interface in DemoApplication.java

<u>CommandLineRunner:</u>

If we need to execute some custom code, just before Spring Boot application start up, we can make that happened with CommandLineRunner. I.e. Spring boot provided CommandLineRunner Interface to run specific pieces of code when an application is fully started. When we want to execute some piece of code exactly before the application startup completes, we can use it.

```
package com.example.demo;

import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.autoconfigure.domain.EntityScan;

@SpringBootApplication
@EntityScan("com.model")
public class DemoApplication implements CommandLineRunner{


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

    @Override
    public void run(String... arg0) throws Exception {

    }

}
```

Here CommandLineRunner interface is having a method

public void run(String.. arg0) throws Exception{
}

Where we can write our custom piece of code which we wants to execute just before the application start up.

Step-5: autowired the Custom Repository I.e. PatientRepository in DemoApplication.java and create the patient object and store it in H2 db.

```
package com.example.demo;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.autoconfigure.domain.EntityScan;

import com.model.Patient;

@SpringBootApplication
@EntityScan("com.model")
public class DemoApplication implements CommandLineRunner{

    @Autowired
    PatientRepository patService;

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

    @Override
    public void run(String... arg0) throws Exception {
        Patient p = new Patient();
        p.setPatName("john124");
        p = patService.save(p);
        System.out.println(p.getPatId());
    }


}
```

Step-5: when we run the application we could see the auto generated id value in the console for the successfully created patient.

```
Hibernate: insert into patient (pat_id, pat_name) values (null, ?)
1
2017-09-15 00:15:29.362  INFO 3156 --- [           main] com.example.demo.DemoApplication         : Started DemoApplication in 4.171 seconds (JVM running for 4.818)
```

DataSource Configuration in Spring Boot

When we add default database I.e. H2 dependency in pom.xml file, as @EnableAutoConfiguration magic annotation been configured, as per convention over configuration approach, spring boot automatically establish the database connection when application starts. In order to connect to this data base, spring boot by default will uses the “org.apache.tomcat.jdbc.pool.DataSource”.

```
D:\trainings\dotridge\demo\target>java -jar demo-0.0.1-SNAPSHOT.jar --debug > test.txt
```

```
DataSourceAutoConfiguration matched:
   - @ConditionalOnClass found required classes 'javax.sql.DataSource', 'org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType';
   @ConditionalOnMissingClass did not find unwanted class (OnClassCondition)

DataSourceAutoConfiguration#dataSourceInitializer matched:
   - @ConditionalOnMissingBean (types: org.springframework.boot.autoconfigure.jdbc.DataSourceInitializer; SearchStrategy: all) did not find any
   beans (OnBeanCondition)

DataSourceAutoConfiguration.PooledDataSourceConfiguration matched:
   - AnyNestedCondition 1 matched 1 did not; NestedCondition on DataSourceAutoConfiguration.PooledDataSourceCondition.PooledDataSourceAvailable
   PooledDataSource found supported DataSource; NestedCondition on DataSourceAutoConfiguration.PooledDataSourceCondition.ExplicitType
   @ConditionalOnProperty (spring.datasource.type) did not find property 'type' (DataSourceAutoConfiguration.PooledDataSourceCondition)
   - @ConditionalOnMissingBean (types: javax.sql.DataSource,javax.sql.XADataSource; SearchStrategy: all) did not find any beans (OnBeanCondition)

DataSourceConfiguration.Tomcat matched:
   - @ConditionalOnClass found required class 'org.apache.tomcat.jdbc.pool.DataSource'; @ConditionalOnMissingClass did not find unwanted class
   (OnClassCondition)
   - @ConditionalOnProperty (spring.datasource.type=org.apache.tomcat.jdbc.pool.DataSource) matched (OnPropertyCondition)

DataSourcePoolMetadataProvidersConfiguration.TomcatDataSourcePoolMetadataProviderConfiguration matched:
   - @ConditionalOnClass found required class 'org.apache.tomcat.jdbc.pool.DataSource'; @ConditionalOnMissingClass did not find unwanted class
   (OnClassCondition)

DataSourceTransactionManagerAutoConfiguration matched:
   - @ConditionalOnClass found required classes 'org.springframework.jdbc.core.JdbcTemplate',
   'org.springframework.transaction.PlatformTransactionManager'; @ConditionalOnMissingClass did not find unwanted class (OnClassCondition)

DataSourceTransactionManagerAutoConfiguration.DataSourceTransactionManagerConfiguration matched:
   - @ConditionalOnSingleCandidate (types: javax.sql.DataSource; SearchStrategy: all) found a primary bean from beans 'dataSource' (OnBeanCondition)
```

Preventing Errors on Dead DB Connections

The above configured data source usually access the database with the connection pool. If our code needs to send a query, it gets the connection from the pool and when it is done it again gives back the connection to the pool. However what can happen due to network issues, DB timeouts etc. Is that the connection got stale. If the connection stale, it is essentially dead and can not be used. Typically the default configured pool doesn't clean them up and the next time when we use a connection we might get a chance to get that broken one. Hence we need to tell the pool to check each connection before using it. To enable this, add the following properties in application.properties /.yml file as below:

```
spring.datasource.tomcat.test-on-borrow=true
spring.datasource.tomcat.validation-query=SELECT 1
spring.datasource.tomcat.initial-size=20
spring.datasource.tomcat.max-active=25
```

Here,

spring.datasource.tomcat.test-on-borrow enables the check for valid connections

spring.datasource.tomcat.validation-query defines a query for the check I.e. Test Connection query.

Note:

This check comes with a drawback I.e. a minor loss in performance but avoid connection pooling issues greatly.

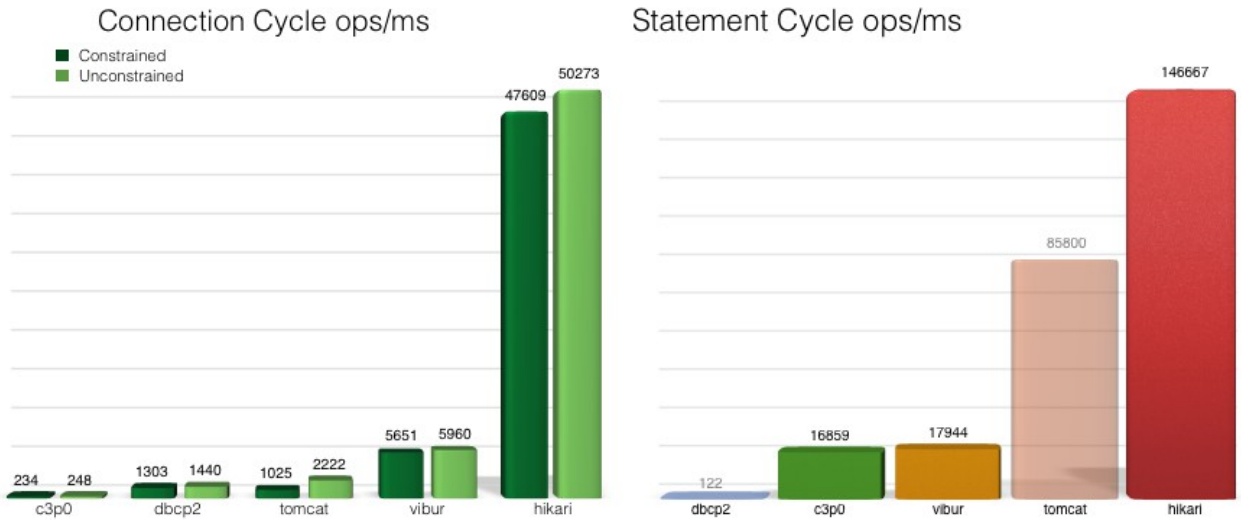
HikariCP Connection Pooling:

HikariCP is a reliable, high-performance JDBC connection pool.

Connection Pool:

a connection pool is a cache of database connections maintained so that the connections can be reused when future requests to the database are required. Connection pools are used to enhance the performance of executing commands on a database.

HikariCP is world first, most perfomace connection pool.



Connection Pool is the buffer or temporary memory area, where we can maintain an equal set of connection objects.

Connection pooling:

The process of maintaining an equal set of connection objects meaning defining how many objects will live, how much time they will active and how minimum time they will sleep etc.

The generic advantage of connection pooling is an un-used objects will be retained back to the pool and will be available for the next request. Hence the maximum number of defined connection objects will only be re-used at any point of time. Hence we could see a great performance improvement over repeated jdbc query calls.

Before adding any connection pooling other than tomcat-jdbc connection pool, we have to exclude it from jdbc starter as shown below:

```
<!-- exclude tomcat jdbc connection pool -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.apache.tomcat</groupId>
            <artifactId>tomcat-jdbc</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

Now as default tomcat-jdbc is excluded spring boot will pick the HikariCP automatically, if it is been configured in pom.xml file as a dependency.

```
<!-- Now tomcat-jdbc excluded, Spring Boot will use HikariCP automatically -->
<dependency>
    <groupId>com.zaxxer</groupId>
    <artifactId>HikariCP</artifactId>
</dependency>
```

Example:

Step-1: Configure the H2 db console enabling, view log statement and update table statements in application.properties file as shown below:

```
spring.h2.console.enabled=true
spring.h2.console.path=/console

spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=update
```

Step-2: Create the domain using JPA as shown below:

```
package com.example.demo.model;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Student {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Integer studentId;
    private String firstName;
    private String lastName;
    private String email;

    public Integer getStudentId() {
        return studentId;
    }

    public void setStudentId(Integer studentId) {
        this.studentId = studentId;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
```

Step-3: Create the Spring Data Crud Repository as shown below:

```
package com.example.demo.repository;

import org.springframework.data.repository.CrudRepository;

import com.example.demo.model.Student;

public interface StudentRepository extends CrudRepository<Student, Integer> {

}
```

Implementing the Service Layer

Step-4: Create the Service Layer class shown below:

```
package com.example.demo.services;

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

import com.example.demo.model.Student;
import com.example.demo.repository.StudentRepository;

@Service
public class StudentService {

    @Autowired
    private StudentRepository rep;

    //@Transactional
    public Student saveStudent(Student st) {
        return rep.save(st);
    }
}
```

Step-5: Autowired the Service Layer class in DemoApplication1.java file as shown below:

```java
package com.example.demo;

import javax.sql.DataSource;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

import com.example.demo.model.Student;
import com.example.demo.services.StudentService;

@SpringBootApplication
public class Demo1Application implements CommandLineRunner{

    @Autowired
    DataSource ds;
    @Autowired
    StudentService stRepo;

    public static void main(String[] args) {
        SpringApplication.run(Demo1Application.class, args);
    }

    @Override
    public void run(String... arg0) throws Exception {
        System.out.println("current data source is:\t"+ds);
        Student st = new Student();
        st.setEmail("john@gmail.com");
        st.setFirstName("john");
        stRepo.saveStudent(st);
        System.out.println("student ID is:\t"+st.getStudentId());
        System.out.println("student Email is:\t"+st.getEmail());
    }
}
```

Note:

In the above Demo1Application.java file, we have autowired the **"javax.sql.DataSource"** , because of just to know which connection pool boot is using when connecting to H2 database, as we have excluded the default "tomcat-jdbc" connection pooling.

Step-6: Now run the application and we could see the below log on console:

```
current data source is: HikariDataSource (HikariPool-1)
Hibernate: insert into student (student_id, email, first_name, last_name) values (null, ?, ?, ?)
student ID is:  1
student Email is:       john@gmail.com
2017-09-15 15:57:56.807  INFO 8248 --- [           main] com.example.demo.Demo1Application        : Started Demo1Application in 3.665 seconds (JVM running for 4.273)
```

Note:

1) In the above HikariCP connection pool, we haven't configure the custom connection pooling mechanism like how many objects we wants to live, pool size and time out. We left it to spring boot to configure its own way of pooling. Now if we wants to customize, then add the following properties into application.properties file as shown below:

```
# HikariCP settings
# spring.datasource.hikari.*

#60 sec
spring.datasource.hikari.connection-timeout=60000
# max 5
spring.datasource.hikari.maximum-pool-size=5

# minimum idle for 10 sec
spring.datasource.hikari.minimum-idle=10000
```

2) As We have used the default H2 database in the above example, we haven't added any driver-class-name, url, user name and password. Because those were automatically taken by spring boot in the run time.

```
16:58:33.818  INFO 9560 --- [          main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Started.
16:58:33.927  INFO 9560 --- [          main] j.LocalContainerEntityManagerFactoryBean : Building JPA container EntityManagerFactory for persistence unit 'default'
16:58:33.940  INFO 9560 --- [          main] o.hibernate.jpa.internal.util.LogHelper  : HHH000204: Processing PersistenceUnitInfo [
me: default
.]
16:58:33.994  INFO 9560 --- [          main] org.hibernate.Version                    : HHH000412: Hibernate Core {5.0.12.Final}
16:58:33.995  INFO 9560 --- [          main] org.hibernate.cfg.Environment            : HHH000206: hibernate.properties not found
16:58:33.996  INFO 9560 --- [          main] org.hibernate.cfg.Environment            : HHH000021: Bytecode provider name : javassist
16:58:34.032  INFO 9560 --- [          main] o.hibernate.annotations.common.Version   : HCANN000001: Hibernate Commons Annotations {5.0.1.Final}
16:58:34.127  INFO 9560 --- [          main] org.hibernate.dialect.Dialect            : HHH000400: Using dialect: org.hibernate.dialect.H2Dialect
16:58:34.527  INFO 9560 --- [          main] org.hibernate.tool.hbm2ddl.SchemaUpdate  : HHH000228: Running hbm2ddl schema update
```

In the above screen shot we could see, hibernate settings like Dialect,version and schema updates etc.

This is how? Are we using hibernate so far?

No. This is because of, the spring-data-jpa will use the the JPA2.1 specification which is supporting the following features:

- Converters - allowing custom code conversions between database and object types.
- Criteria Update/Delete - allows bulk updates and deletes through the Criteria API.
- Entity Graphs - allow partial or specified fetching or merging of objects.
- JPQL/Criteria enhancements - arithmetic sub-queries, generic database functions, join ON clause, TREAT option.
- Schema Generation
- Stored Procedures - allows queries to be defined for database stored procedures.

And the Vendors supporting the JPA2.1 are:

- DataNucleus
- EclipseLink
- Hibernate

So When we use, spring-data-jpa, then boot automatically configures the JPA2.1 features with the default vendor as Hibernate.

3) As We have used the default H2 database in the above example, we haven't added any driver-class-name, url, user name and password. Because those were automatically taken by spring boot in the run time. But if we wants to use other database like MySQL, we have to do the following configuratioons:

A) add the my-sql-connector in pom.xml file as shown below:

```
<!-- For MySQL -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
```

B) add the driver-class name, url, username and password as shown bellow:

```
# MySQL DB configurations
spring.datasource.driver-class-name= com.mysql.jdbc.Driver
spring.datasource.url= jdbc:mysql://localhost:3306/testdb
spring.datasource.username=root
spring.datasource.password=root
```

On the basis of given data source URL, spring boot can automatically identify data source driver class. So we need not to configure diver class.

Spring Boot with Hibernate and Custom database I.e. My SQL

The data source properties starting with **spring.datasource.*** will automatically be read by spring boot JPA. To change the hibernate properties we will use prefix **spring.jpa.properties.*** with hibernate property name. On the basis of given data source URL, spring boot can automatically identify data source driver class. So we need not to configure diver class.

Step-1: Configure the hibernate properties as shown below:

```
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5Dialect
spring.jpa.properties.hibernate.format_sql = true
spring.jpa.properties.hibernate.current_session_context_class=org.springframework.orm.hibernate4.SpringSessionContext
```

Here "`spring.jpa.properties.hibernate.current_session_context_class=org.springframework.orm.hibernate4.SpringSessionContext`". This property is an Implementation of Hibernate 3.1's CurrentSessionContext interface which delegates to Spring's SessionFactoryUtils for providing a Spring-managed current Session.This CurrentSessionContext implementation can also be specified in custom SessionFactory setup through the "hibernate.current_session_context_class" property, with the fully qualified name of this class as value.

Note:

If we wants to use Hibernate5 features as SpringSessionContext, then change the above property as "`spring.jpa.properties.hibernate.current_session_context_class=org.springframework.orm.hibernate5.SpringSessionContext`"

Step-2: Define HibernateJpaSessionFactoryBean as a bean in Demo1Application.java as shown below:

```
import org.springframework.context.annotation.Bean;
import org.springframework.orm.jpa.vendor.HibernateJpaSessionFactoryBean;

import com.example.demo.model.Student;
import com.example.demo.services.StudentService;

@SpringBootApplication
public class Demo1Application implements CommandLineRunner{
    @Autowired
    DataSource ds;
    @Autowired
    StudentService stRepo;

    public static void main(String[] args) {
        SpringApplication.run(Demo1Application.class, args);
    }

    @Bean
    public HibernateJpaSessionFactoryBean sessionFactory() {
        return new HibernateJpaSessionFactoryBean();
    }

    @Override
    public void run(String... arg0) throws Exception {
        System.out.println("current data source is:\t"+ds);
        Student st = new Student();
        st.setEmail("john@gmail.com");
        st.setFirstName("john");
        stRepo.saveStudent(st);
        System.out.println("student ID is:\t"+st.getStudentId());
        System.out.println("student Email is:\t"+st.getEmail());
    }
}
```

Here HIbernateJpaSessionFactoryBean is a Simple FactoryBean that **exposes the underlying SessionFactory behind a Hibernate-backed JPA EntityManagerFactory.**

Primarily available for resolving a SessionFactory by JPA persistence unit name via the "persistenceUnitName" bean property.

Step-3: Autowire the SessionFactory in DAO Impl classes as shown below

```java
package com.example.demo.repository;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;

import com.example.demo.model.Student;

@Repository
public class StudentRepositoryimpl implements StudentRepository {

    @Autowired
    SessionFactory sf;

    @Override
    public Student addStudent(Student st) {
        Session ses = sf.openSession();
        ses.save(st);
        return st;
    }

}
```

Step-4: when we run the application, we could see the log with the record created in database as shown below:

```
2017-09-15 20:16:16.574  INFO 1092 --- [           main] org.hibernate.Version                    : HHH000412: Hibernate Core {5.0.12.Final}
2017-09-15 20:16:16.575  INFO 1092 --- [           main] org.hibernate.cfg.Environment            : HHH000206: hibernate.properties not found
2017-09-15 20:16:16.577  INFO 1092 --- [           main] org.hibernate.cfg.Environment            : HHH000021: Bytecode provider name : javassist
2017-09-15 20:16:16.627  INFO 1092 --- [           main] o.hibernate.annotations.common.Version   : HCANN000001: Hibernate Commons Annotations {5.0.1.Final}
2017-09-15 20:16:16.745  INFO 1092 --- [           main] org.hibernate.dialect.Dialect            : HHH000400: Using dialect: org.hibernate.dialect.MySQL5Dialect
2017-09-15 20:16:17.277  INFO 1092 --- [           main] org.hibernate.tool.hbm2ddl.SchemaUpdate  : HHH000228: Running hbm2ddl schema update
2017-09-15 20:16:17.289  INFO 1092 --- [           main] rmationExtractorJdbcDatabaseMetaDataImpl : HHH000262: Table not found: student
2017-09-15 20:16:17.290  INFO 1092 --- [           main] rmationExtractorJdbcDatabaseMetaDataImpl : HHH000262: Table not found: student
2017-09-15 20:16:17.442  INFO 1092 --- [           main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2017-09-15 20:16:17.917  INFO 1092 --- [           main] s.w.s.m.m.a.RequestMappingHandlerAdapter : Looking for @ControllerAdvice: org.springframework.boot.context.emb
2017-09-15 20:16:17.975  INFO 1092 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/error]}" onto public org.springframework.http.ResponseEr
2017-09-15 20:16:17.977  INFO 1092 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/error],produces=[text/html]}" onto public org.springfram
2017-09-15 20:16:18.004  INFO 1092 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/webjars/**] onto handler of type [class org.sprir
2017-09-15 20:16:18.004  INFO 1092 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/**] onto handler of type [class org.springframewo
2017-09-15 20:16:18.038  INFO 1092 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/**/favicon.ico] onto handler of type [class org.s
2017-09-15 20:16:18.296  INFO 1092 --- [           main] o.s.j.e.a.AnnotationMBeanExporter        : Registering beans for JMX exposure on startup
2017-09-15 20:16:18.298  INFO 1092 --- [           main] o.s.j.e.a.AnnotationMBeanExporter        : Bean with name 'dataSource' has been autodetected for JMX exposure
2017-09-15 20:16:18.303  INFO 1092 --- [           main] o.s.j.e.a.AnnotationMBeanExporter        : Located MBean 'dataSource': registering with JMX server as MBean [c
2017-09-15 20:16:18.373  INFO 1092 --- [           main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
current data source is: HikariDataSource (HikariPool-1)
Hibernate:
    insert
    into
        student
        (email, first_name, last_name)
    values
        (?, ?, ?)
student ID is:  1
student Email is:       john@gmail.com
2017-09-15 20:16:18.470  INFO 1092 --- [           main] com.example.demo.Demo1Application        : Started Demo1Application in 4.166 seconds (JVM running for 4.817)
```

Transaction Management

When working with complex models or different back ends, we usually need some transaction handling. So if anything goes wrong in once backend , we can roll back our data in all the backends. The Spring

Family also provides a module for transaction management I.e. **Spring Transaction Management.** It is used in Spring Data, and we can enable basic version in two simple steps.

Step-1: annotate our service/dao method with @Transactional annotation as shown below:

```
package com.example.demo.services;

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

import com.example.demo.model.Student;
import com.example.demo.repository.StudentRepository;

@Service
public class StudentService {

    @Autowired
    private StudentRepository rep;

    @Transactional
    public Student saveStudent(Student st) {
        return rep.addStudent(st);

    }
}
```

Step-2: Enable the transaction management in the Spring Boot Configuration by adding the **@EnableTransactionManagement** annotation to Demo1Application.java as shown below:

```java
import org.springframework.context.annotation.Bean;
import org.springframework.orm.jpa.vendor.HibernateJpaSessionFactoryBean;
import org.springframework.transaction.annotation.EnableTransactionManagement;

import com.example.demo.model.Student;
import com.example.demo.services.StudentService;

@SpringBootApplication
@EnableTransactionManagement
public class Demo1Application implements CommandLineRunner{
    @Autowired
    DataSource ds;
    @Autowired
    StudentService stRepo;

    public static void main(String[] args) {
        SpringApplication.run(Demo1Application.class, args);
    }

    @Bean
    public HibernateJpaSessionFactoryBean sessionFactory() {
        return new HibernateJpaSessionFactoryBean();
    }

    @Override
    public void run(String... arg0) throws Exception {
        System.out.println("current data source is:\t"+ds);
        Student st = new Student();
        st.setEmail("john@gmail.com");
        st.setFirstName("john");
        stRepo.saveStudent(st);
        System.out.println("student ID is:\t"+st.getStudentId());
        System.out.println("student Email is:\t"+st.getEmail());
    }
```

Step-3: If the transaction is enabled, then we can open a current session in daoimpl rather then every time opening a new session as shown below:

```java
package com.example.demo.repository;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;

import com.example.demo.model.Student;

@Repository
public class StudentRepositoryimpl implements StudentRepository {

    @Autowired
    SessionFactory sf;

    @Override
    public Student addStudent(Student st) {
        //Session ses = sf.openSession();
        Session ses = sf.getCurrentSession();
        ses.save(st);
        return st;
    }

}
```

Step-4: No when we run the application we could see the log as below

```
current data source is: HikariDataSource (HikariPool-1)
Hibernate:
    insert
    into
        student
        (email, first_name, last_name)
    values
        (?, ?, ?)
student ID is:  2
student Email is:       john@gmail.com
2017-09-15 21:02:51.280  INFO 8560 --- [          main] com.example.demo.Demo1Application        : Started Demo1Application in 3.659 seconds (JVM running for 4.299)
```

Logging

By Default Spring Boot uses Logback for logging. It is also set up to enable routing from common java logging libraries like java util Logging, Commons Logging, Log4j or SLF4J.

In the above Demo application, lets use Slf4j.

To Obtain a logger instance, we call getLogger of the **org.slf4j.LoggerFactory.** The method is static and we can create a constant in our class I.e. StudentService.

```java
package com.example.demo.services;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import com.example.demo.model.Student;
import com.example.demo.repository.StudentRepository;

@Service
public class StudentService {

    private static final Logger LOGGER = LoggerFactory.getLogger(StudentService.class);

    @Autowired
    private StudentRepository rep;

    @Transactional
    public Student saveStudent(Student st) {
        LOGGER.info("i am in saveStudent() method of StudentService");
        return rep.addStudent(st);

    }
}
```

Now we can use **info, debug** or any of the other logging methods. When we run the application we can see the log on the console as below:

```
current data source is: HikariDataSource (HikariPool-1)
2017-09-15 22:40:38.591  INFO 9512 --- [           main] c.example.demo.services.StudentService   : i am in saveStudent() method of StudentService
```

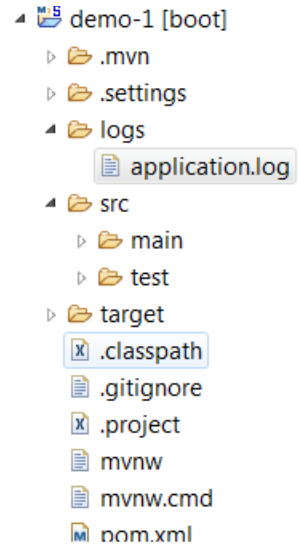
By default our logging message are printed on the console. This is not suitable for running the MicroServices later. So we are going to change it to the file based logging.

Logging to a File:

Spring Boot offers, two properties to enable file logging. They can work together too.

```
logging.file=logs/application.log
```

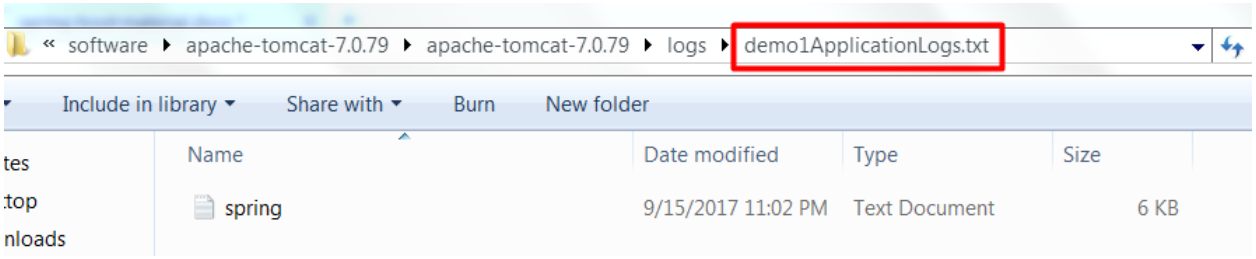
Now when we run the application, we could see a folder “logs” been created and inside in it application.log is created which holds all the application logs.



Here **logging.file** property specifies a file to write in. It can be a relative or absolute path. **It by default creates the file in current directory.**

```
logging.path=${catalina.home}/logs/demo1ApplicationLogs.txt
```

Now when we run the application, we could see a folder "**demo1ApplicationLogs.txt**" been created and inside in it **spring.log** is created which holds all the application logs.



Here logging.path property specifies a directory where to store the log file. If logging.file contains a relative path or just a file name. It will be logging.path if we combine use both the properties. If logging.file is not present, then log file is named as **spring.log**

Changing the Log level

Spring Boot provides a way to modify the log level with properties set in **application.properties** file. They are prefixed by **logging.level** and the value is the one of these log level TRACE,DEBUG,INFO,WARN,ERROR,FATAL,OFF. To set the log level for the root logger use

```
logging.level.root=warn
```

We can even filter the logs by application wise, for instance, if we want classes from the spring framework to only report errors and from the Demo1Application only to report INFO messages then configure the below properties:

```
logging.level.org.org.springframework=error
logging.level.com.example.demo.Demo1Application=info
```

If we wants to define our own custom logging pattern, that we can define with property as below:

```
# Logging pattern for file
logging.pattern.file= %d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg%n
```

Now when we run the application, we could see the log with the above pattern as:

```
2017-09-16 00:44:46 [http-nio-8080-exec-1] INFO  c.e.demo.controllers.Demo1Controller - getAllStudents()--->
2017-09-16 00:44:46 [http-nio-8080-exec-1] INFO  o.h.h.i.QueryTranslatorFactoryInitiator - HHH000397: Using ASTQueryTranslatorFactory
2017-09-16 00:44:46 [http-nio-8080-exec-1] INFO  c.e.demo.controllers.Demo1Controller - students list is-->[com.example.demo.model.Student@27689c6f,
```

HealthChecks and Metrics

Production readiness is about looking beyond functional requirements and ensuring our application can be properly managed and monitored in production. Some key things to consider when thinking about production readiness are:

- ✓ Health checks
- ✓ Viewing application configuration - application properties, Environment variables etc.
- ✓ Viewing and altering log configuration
- ✓ Viewing application metrics - JVM, classloader, threading and garbage collection.
- ✓ Audibility of key application events

Spring provides all of this functionality out of the box via **Spring Boot Actuator, a sub project of Spring Boot.** A range of RESTful management and monitoring endpoints are provided so we don't have to implement these features each time we build an application. To enable these endpoints, simply add the Actuator starter POM to the project.

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

Endpoints

Actuator endpoints allow us to monitor and interact with our application. Spring Boot includes a number of built-in endpoints and we can also add our own. Most endpoints are sensitive – meaning they're not fully public – while a **/health and /info** are public by default.
Here's some of the most common endpoints Boot provides out of the box:

1) **/actuator:** Provides a hypermedia-based "discovery page" for the other endpoints. Requires Spring HATEOAS to be on the classpath. Sensitive by Default.
2) **/autoconfig-** Displays an auto-configuration report showing all auto-configuration candidates and the reason why they 'were' or 'were not' applied. Sensitive by Default.
3) **/beans-** Displays a complete list of all the Spring beans in our application. Sensitive by Default.

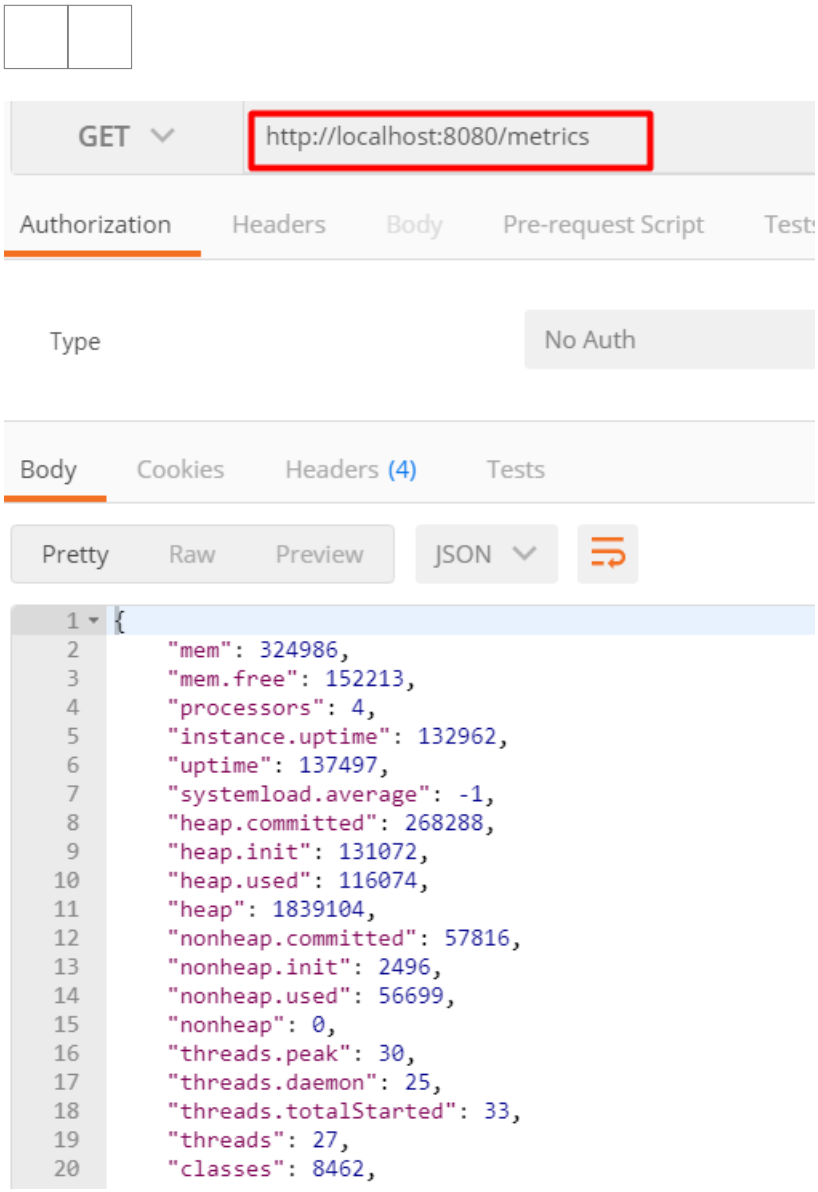
4) **/configprops-** This endpoint shows configuration properties used by our application. Sensitive by Default.
5) **/dump-** Performs a thread dump. Sensitive by Default.
6) **/env-** Exposes spring's properties from the configurations. Sensitive by Default.
7) **/health** – Shows application health information (a simple 'status' when accessed over an unauthenticated connection or full message details when authenticated). It is sensitive by default.
8) **/info** – Displays arbitrary application info. Not sensitive by default.
9) **/metrics** – Shows 'metrics' information for the current application. It is also sensitive by default.
10) **/mappings-** Displays a list of all @RequestMapping paths. Sensitive by Default.
11) **/shutdown-** This endpoint allows to shutdown the application. This is not enabled by default. Sensitive by Default.
12) **/trace** – Displays trace information (by default the last few HTTP requests). Sensitive by Default.
13) **/loggers**– Provides access to the configured log files (This feature supported since Spring Boot 1.3.0). Sensitive by Default.
14) **/flyway-** This endpoint provides the details of any flyway database migrations have been applied. Sensitive by Default.
15) **/liquibase-** This endpoint provides the details of any liquibase database migrations have been applied. Sensitive by Default.

Endpoint Security

By default most Actuator endpoints are considered sensitive. Only the **health and info endpoints are non sensitive (by default) and can be accessed without authentication.** All other endpoints must be accessed using HTTP basic authentication. The sensitivity of individual endpoints can be configured in application.properties. For example, we could change the default security setting and make the metrics endpoint accessible without basic authentication as below:

```
endpoints.metrics.sensitive=false
```

Now when we access the metrics end point as shown below, all the application metrics displayed as shown here:



Its also possible to disable security on all Actuator endpoints as follows.

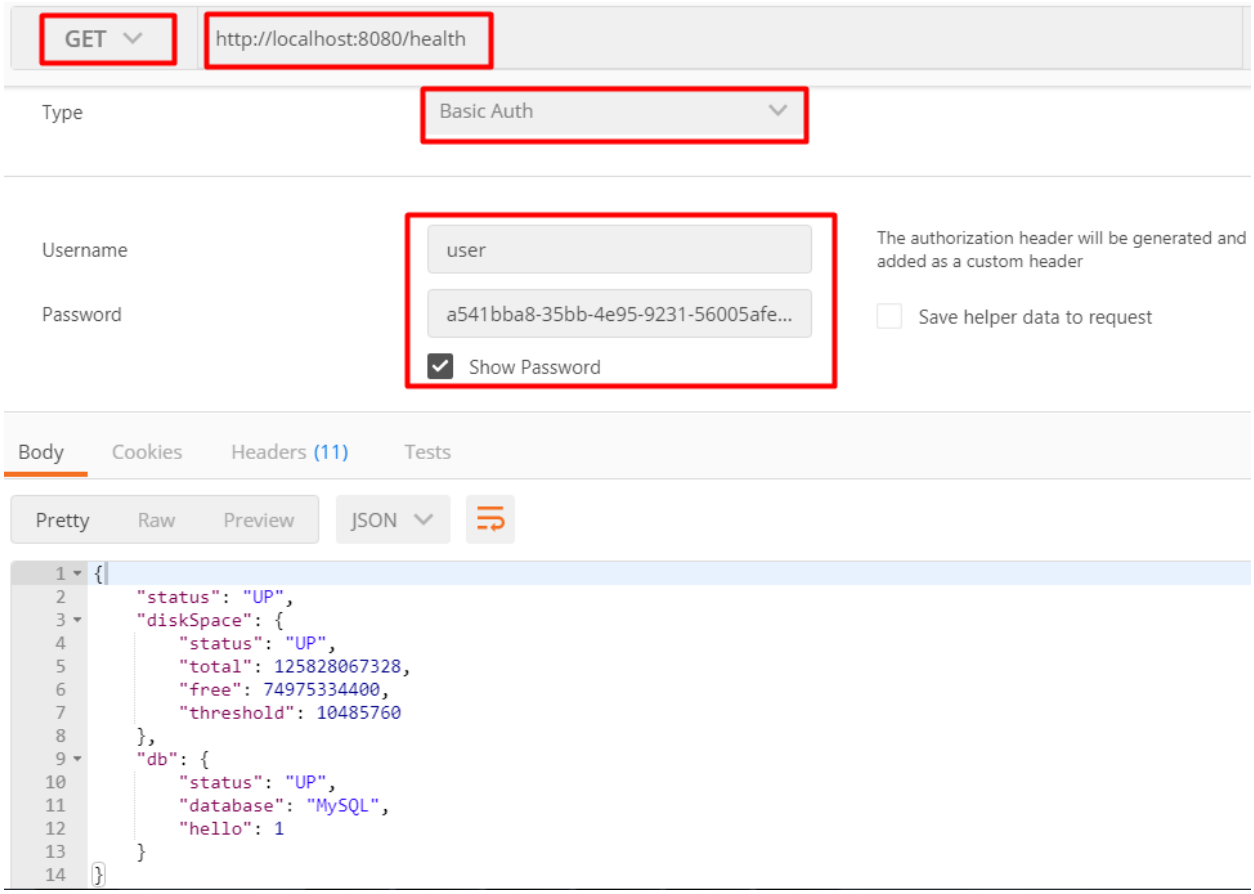
```
management.security.enabled=false
```

Actuator endpoints expose lots of data about the internals of our application so we need to be very careful when disabling security. If our application sits behind a firewall it may be acceptable to disable security, but if the application is public facing then securing the Actuator endpoints is an absolute must. It would be to use basic auth in all cases. Even if our application is sitting behind a firewall it does no harm at all to have an extra layer of security in place.

Health Checks

A health check is an endpoint that returns a message describing the health of an application. An application is considered healthy if its capable of successfully servicing client requests. To do that its core components or dependencies should be up and running. Consider an application that interacts with a database and a JMS message broker.

The health check for such an application might consist of a simple query against the database and creating a connection with the JMS broker. By providing these core components are up and running we can say with a fair degree of confidence that our application is healthy.
Actuator provides a health check endpoint out of the box, which can be accessed at **/health** using a simple HTTP GET. Using postman for example we can call the health check endpoint as



The JSON response above is made up of ***status, diskSpace* and *db***.
Status: describes the overall health of the application, in this case *UP* indicates that the application is healthy.
diskSpace: describes total disk space, remaining disk space and a minimum threshold. The threshold value is configurable in application.properties using ***management.health.diskspace.threshold.*** If this value is greater than the *free* value, the status of the *diskSpace* check will be *DOWN*.
db: describes the health of the database and is derived by running a simple query. By default Boot will look up the validation query for the specific database engine, for example on MySQL the validation query will be ***select 1 from dual***. We can also override the default query by writing our own health check implementations.

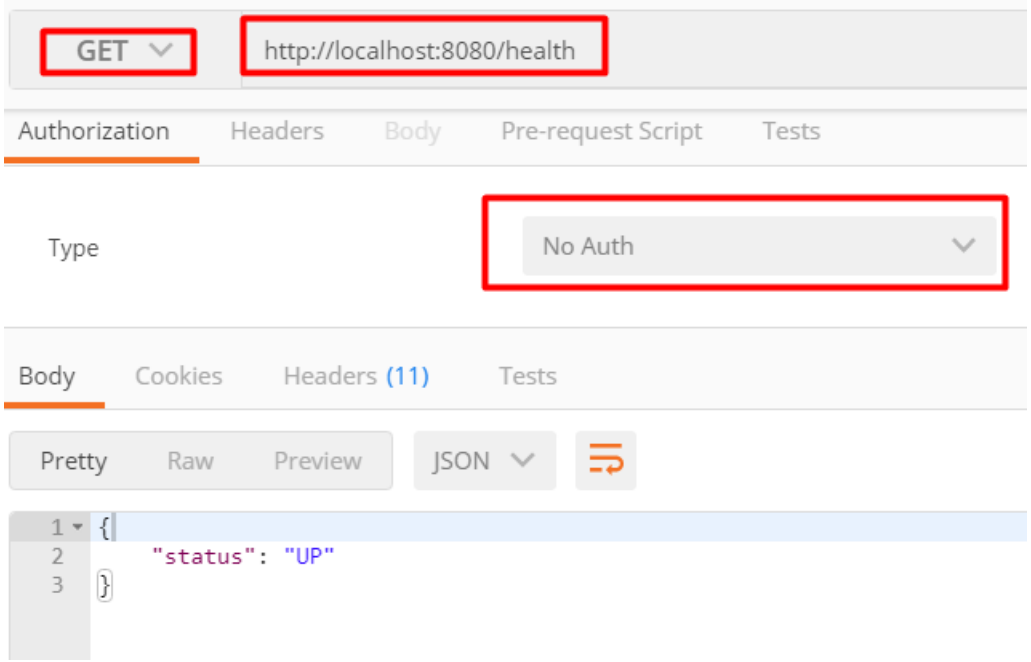
When the status UP is returned, a HTTP 200 response code is used. If the health check returns DOWN the response code is HTTP 503 (Service Unavailable). This is useful for load balancers or monitoring tools that may be calling the health check. They can derive the health status using the HTTP response code rather than the response message, which can be simply logged for information. If the format of the response message happens to change it wont break the tools that are calling the health endpoint.

Automatic Health Checks

The sample response above consists of 2 separate checks, disk space and database. Spring Boot automatically detects components used by an application and runs the appropriate health checks. In the above application, it detects a DataSource and therefore runs the database health check. If a JmsConnectionFactory was configured in the application, it would also run a JMS health check. As well as the Actuator also provides automatic health checks for SMTP, Mongo, Redis, ElasticSearch, RabbitMQ and more. If we would prefer to disable these automatic health checks we can do so by setting **management.health.defaults.enabled=false** in **application.properties.**

Health Check Security

By default when the health check is called without basic authentication, only the status (UP or DOWN) is returned in the response body.



This ensures that sensitive information such as the name of the database engine aren't exposed to the world. When basic authentication is used a detailed JSON response is returned.

Custom Health Checks

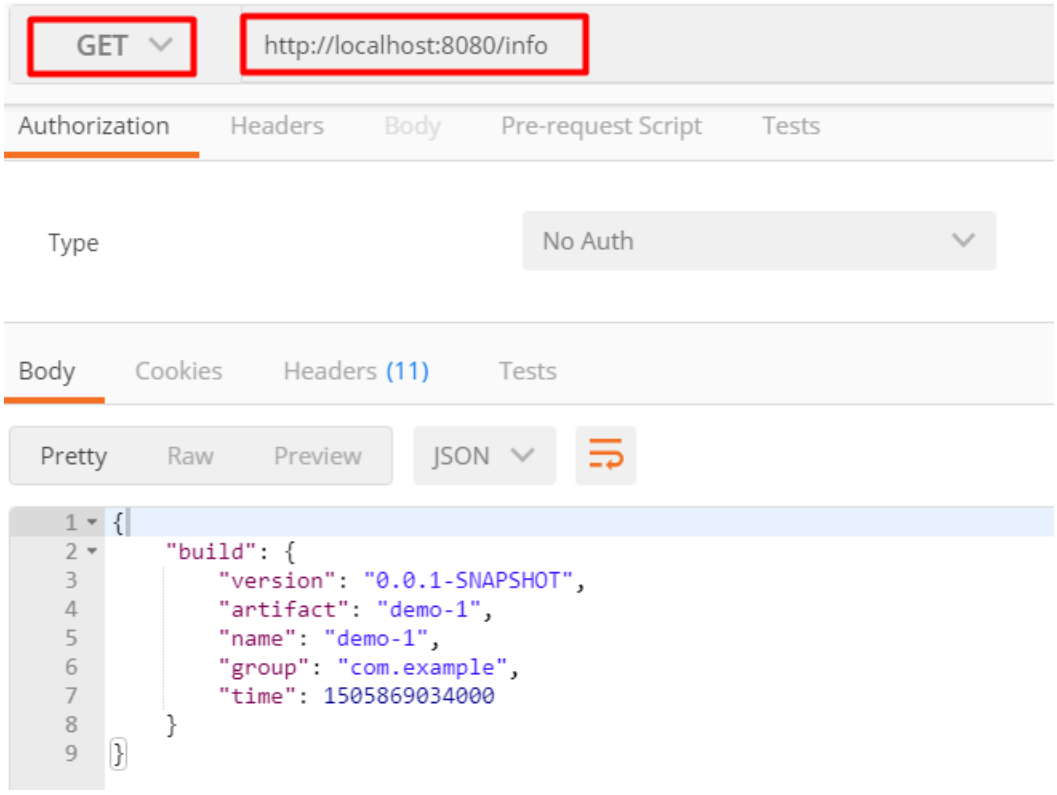
If the standard health checks aren't sufficient and we need something customized, then we can override the default health check with our own implementation. For this Simply create a class that implements HealthIndicator and override the health method as follows.

```java
package com.example.demo.config;

import org.springframework.boot.actuate.health.Health;
import org.springframework.boot.actuate.health.HealthIndicator;
import org.springframework.stereotype.Component;

@Component
public class HealthCheck implements HealthIndicator {

    @Override
    public Health health() {

        if (isRemoteServiceUp()) {
            return Health.up().withDetail("remote service", "online").build();
        }

        return Health.down().withDetail("remote service", "offline").build();
    }

    private boolean isRemoteServiceUp() {

        // perform call out to remote service to check if its up
        return true;
    }

}
```

In the example above we have created a dummy method ***isRemoteServiceUp*** to check the status of some fictitious remote service. In reality this method would perform a remote call to check that the service is up and the resulting status used to build a health response.

Info Endpoint

The info endpoint provides general information about our application. The type of information returned can be customized, but one area that is particularly useful is the application versioning information. There's been confusion around the version of a running application. Being able to check application info on a running instance is a great way to confirm that a deployment has gone as expected and the new version of the application is indeed up and running. The info endpoint is can be called as follows

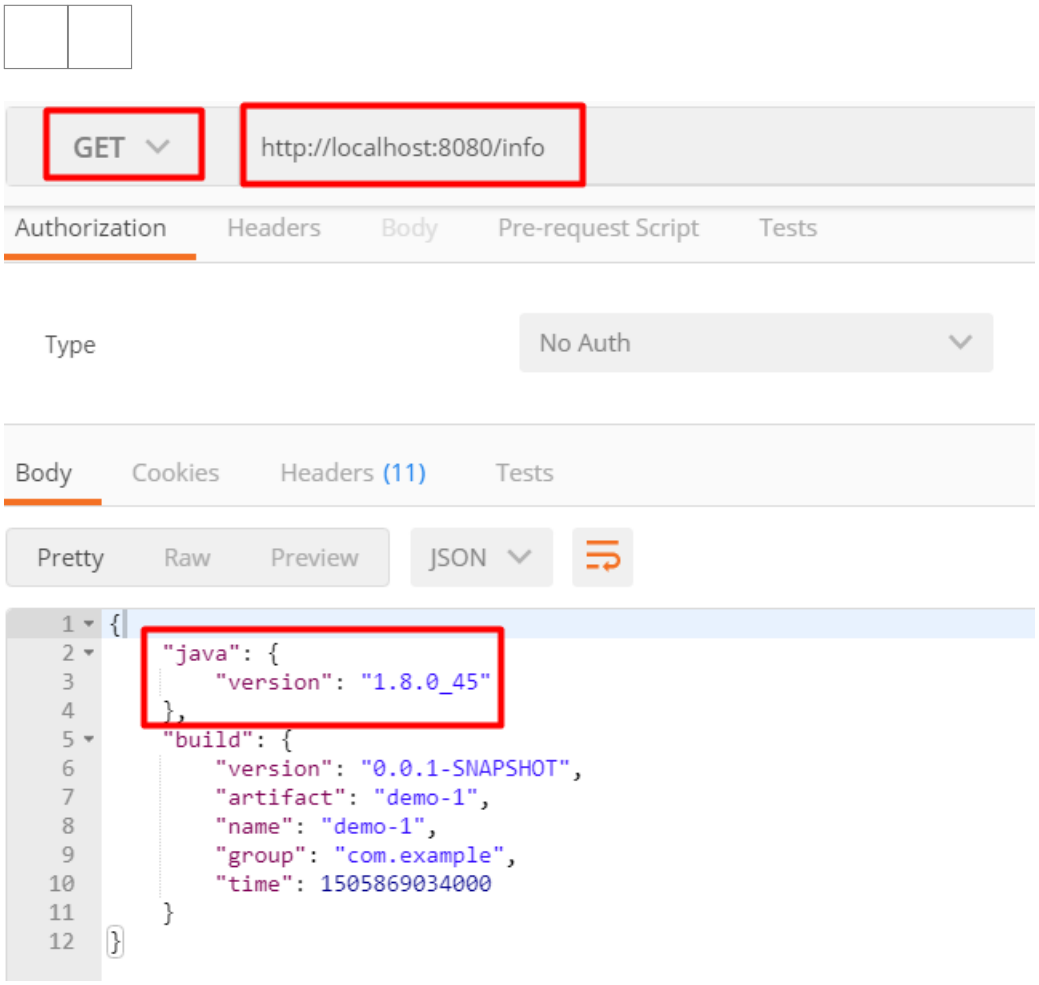


Custom Application Information

We can add custom information to the *info* endpoint response via ***application.properties*** *as shown below:*

```
endpoints.info.id=info
info.java.version=@java.version@
```

Here the @...@ notation allows us to access any value from the maven *project.properties* at build time. Now access the info end point as shown below:

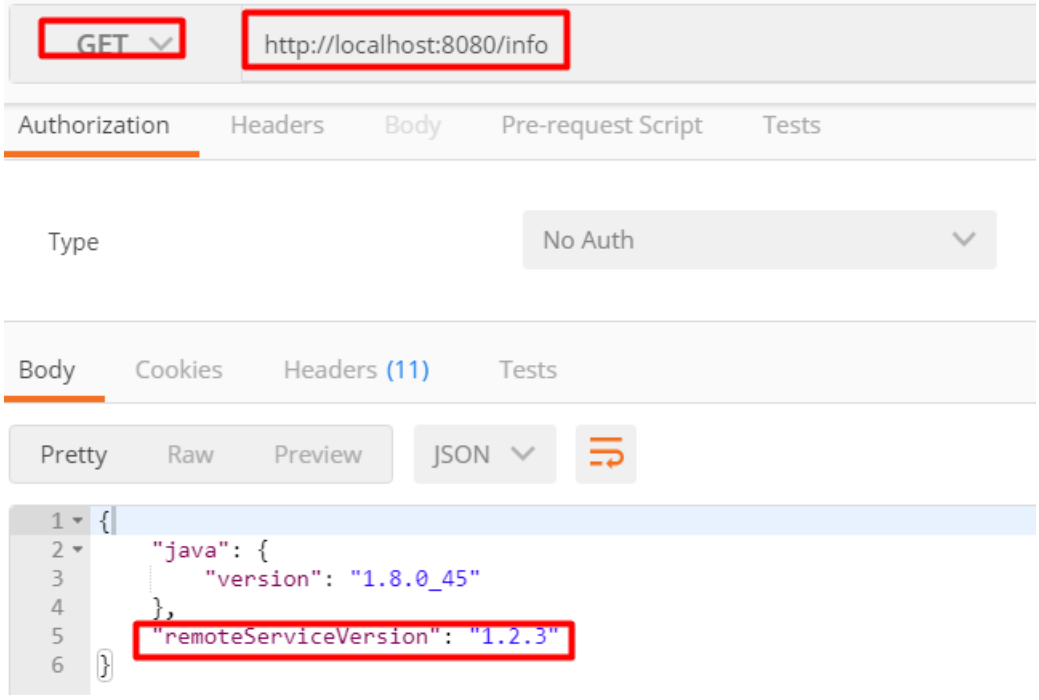


In the response above, *java.version* is populated using the configuration in *application.properties*.

There may be data we want to expose on the info endpoint that isn't available until runtime. We can do this by creating a class that implements the **InfoContributor** interface. This class allows us to add whatever information we want to the info endpoint. The example below exposes the version of some fictitious remote service.

```java
package com.example.demo.config;

import org.springframework.boot.actuate.info.Info.Builder;
import org.springframework.boot.actuate.info.InfoContributor;
import org.springframework.stereotype.Component;

@Component
public class CustomInfoContributor implements InfoContributor {

    @Override
    public void contribute(Builder builder) {
        // add whatever data you want to expose on info endpoint...
        builder.withDetail("remoteServiceVersion", getRemoteServiceVersion());
    }

    // service would check version of some remote service being used
    private String getRemoteServiceVersion() {
        return "1.2.3";
    }

}
```

Now when we access the info end point as shown below, we could see the remoteServiceVersion as below:

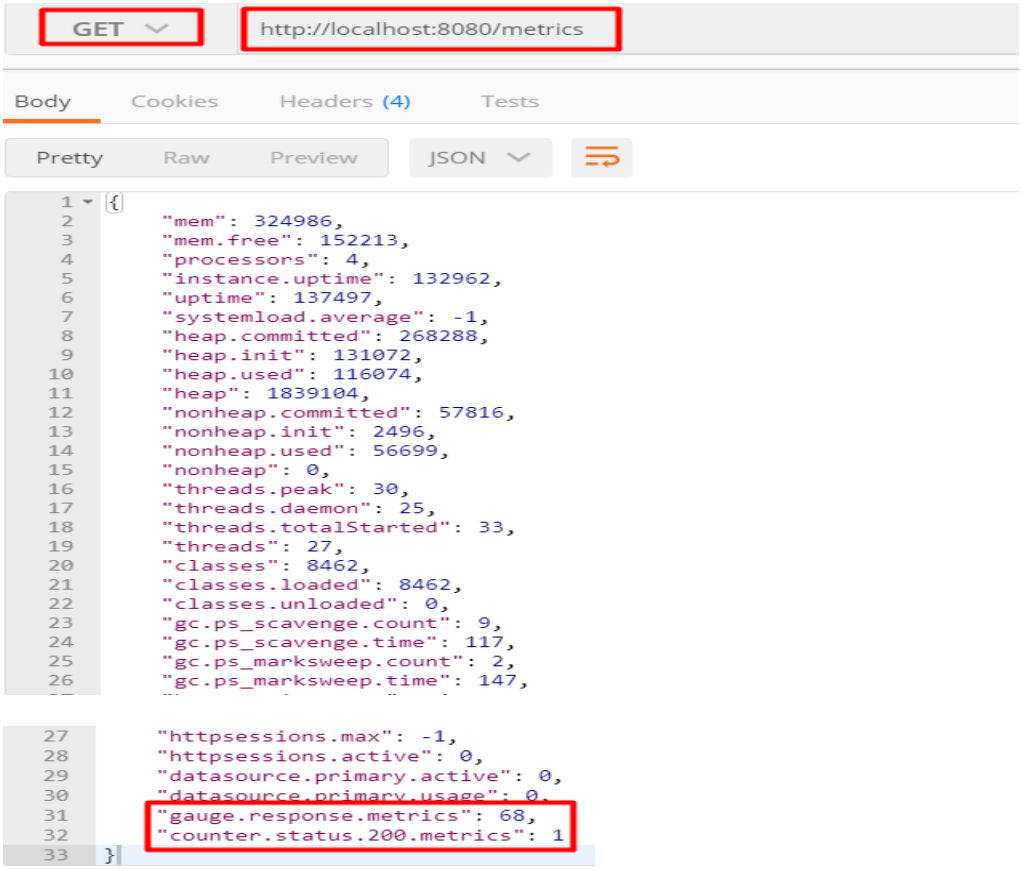


Metrics Endpoint

The *metrics* endpoint exposes a range of application metrics that are extremely useful when it comes to **monitoring and trouble shooting** poorly performing applications. Data exposed by the *metrics*endpoint includes

- System Memory - Total and free system memory in KB
- System Uptime - Server uptime and application context uptime
- JVM Memory Usage - Initial, used and max available heap and non heap size
- Thread Statistics - total thread count, number of threads started and number of daemon threads
- Classloader - total classes loaded, number currently loaded and number unloaded
- Garbage Collector - GC algorithm run, number of times it has run and time taken for last GC
- HTTP Sessions - Current and peak number of HTTP sessions
- Data Source - total number of connections and total active connections
- Endpoints - number of times each endpoint has been called and the last response time for each endpoint

Using *Postman* we can access the *metrics* endpoint as below:

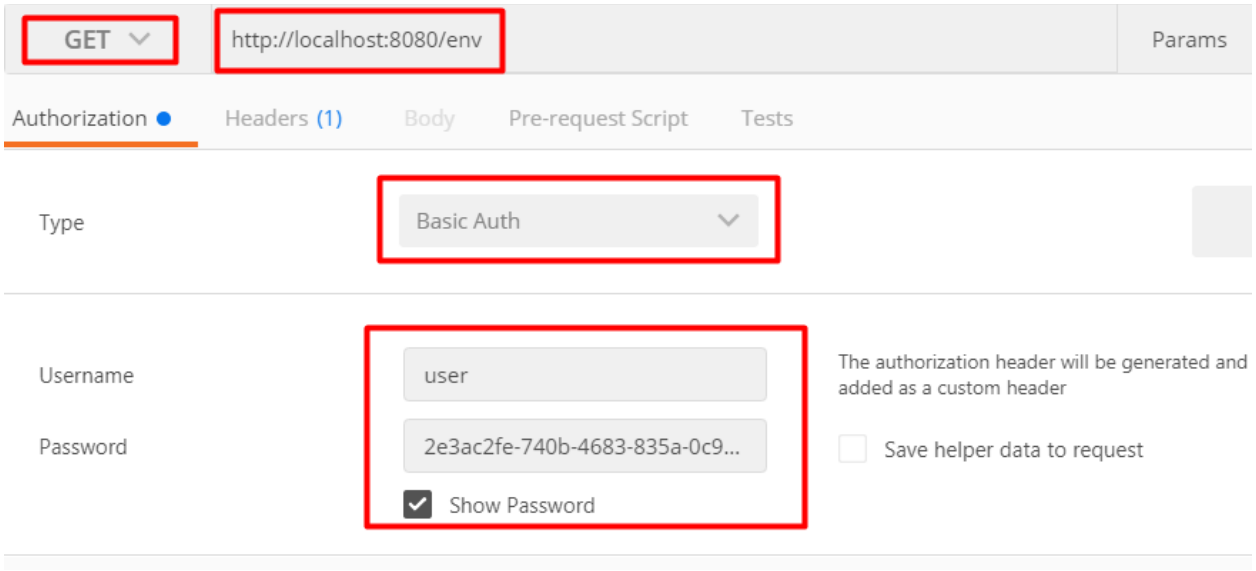


Most of the data in the sample response is fairly intuitive. One section worth describing further is the use of gauge and counter for endpoint metrics. A gauge is used to record a single value, for example on line **31** ***gauge.response.metrics: 68.0*** shows the last response time in milliseconds for the *metrics* endpoint. A *counter* records the increment or decrement of a value, for example on line

32 ***counter.status.200.metrics: 1*** shows the number of times the *metrics* endpoint has been called since the app started.

Env Endpoint

The *env* endpoint exposes data from the application *Environment*, the object that encapsulates all configuration available to the running application. Using *postman* we can access the *env* endpoint as:



Below is a sample response that includes **profiles, server ports, Servlet initialising parameters, system properties, environment variables and application properties**.


```
{
    "profiles": [],
    "server.ports": {
        "local.server.port": 8080
    },
    "commandLineArgs": {
        "spring.output.ansi.enabled": "always"
    },
    "servletContextInitParams": {},
    "systemProperties": {
        "java.vendor": "Oracle Corporation",
        "sun.java.launcher": "SUN_STANDARD",
        "catalina.base": "C:\\Users\\nsanda\\AppData\\Local\\Temp\\tomcat.6155583169280417407.8080",
        "sun.management.compiler": "HotSpot 64-Bit Tiered Compilers",
        "catalina.useNaming": "false",
        "os.name": "Windows 7",
        "sun.boot.class.path": "C:\\Program Files\\Java\\jdk1.8.0_45\\jre\\lib\\resources.jar;C:\\Program Files\\Java\\jdk1.8.0_45\\jre\\lib\\rt
            .jar;C:\\Program Files\\Java\\jdk1.8.0_45\\jre\\lib\\sunrsasign.jar;C:\\Program Files\\Java\\jdk1.8.0_45\\jre\\lib\\jsse.jar;C
            :\\Program Files\\Java\\jdk1.8.0_45\\jre\\lib\\jce.jar;C:\\Program Files\\Java\\jdk1.8.0_45\\jre\\lib\\charsets.jar;C:\\Program
            Files\\Java\\jdk1.8.0_45\\jre\\lib\\jfr.jar;C:\\Program Files\\Java\\jdk1.8.0_45\\jre\\classes",
        "sun.desktop": "windows",
        "spring.application.admin.enabled": "true",
        "com.sun.management.jmxremote": "",
        "java.vm.specification.vendor": "Oracle Corporation",
        "java.runtime.version": "1.8.0_45-b14",
        "spring.liveBeansView.mbeanDomain": "",
        "user.name": "nsanda",
```


```
    "FILE_LOG_PATTERN": "%d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg%n",
    "user.language": "en",
    "sun.boot.library.path": "C:\\Program Files\\Java\\jdk1.8.0_45\\jre\\bin",
    "com.zaxxer.hikari.pool_number": "1",
    "com.sun.management.jmxremote.port": "58001",
    "PID": "4616",
    "java.version": "1.8.0_45",
    "user.timezone": "Asia/Calcutta",
    "sun.arch.data.model": "64",
    "java.endorsed.dirs": "C:\\Program Files\\Java\\jdk1.8.0_45\\jre\\lib\\endorsed",
    "java.rmi.server.randomIDs": "true",
    "sun.cpu.isalist": "amd64",
    "sun.jnu.encoding": "Cp1252",
    "file.encoding.pkg": "sun.io",
    "file.separator": "\\",
    "java.specification.name": "Java Platform API Specification",
    "java.class.version": "52.0",
    "user.country": "US",
    "LOG_FILE": "D:\\software\\apache-tomcat-7.0.79\\apache-tomcat-7.0.79/logs/demo1logs.log",
    "java.home": "C:\\Program Files\\Java\\jdk1.8.0_45\\jre",
    "java.vm.info": "mixed mode",
    "os.version": "6.1",
    "com.sun.management.jmxremote.ssl": "false",
    "path.separator": ";",
    "java.vm.version": "25.45-b02",
    "org.jboss.logging.provider": "slf4j",
```



```
"applicationConfig: [classpath:/application.properties]": {
    "logging.pattern.file": "%d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg%n",
    "spring.datasource.hikari.connection-timeout": "60000",
    "spring.jpa.properties.hibernate.dialect": "org.hibernate.dialect.MySQL5Dialect",
    "logging.level.com.example.demo.controllers.Demo1Controller": "debug",
    "spring.datasource.url": "jdbc:mysql://localhost:3306/testdb",
    "endpoints.info.id": "info",
    "spring.h2.console.path": "/console",
    "spring.datasource.password": "******",
    "spring.datasource.driver-class-name": "com.mysql.jdbc.Driver",
    "spring.jpa.properties.hibernate.format_sql": "true",
    "spring.application.name": "mystudent",
    "spring.jpa.properties.hibernate.current_session_context_class": "org.springframework.orm.hibernate4.SpringSessionContext",
    "spring.h2.console.enabled": "true",
    "spring.datasource.username": "root",
    "logging.file": "C:\\Users\\nsanda\\AppData\\Local\\Temp\\tomcat.6155583169280417407.8080/logs/demo1logs.log",
    "info.java.version": "1.8.0_45",
    "spring.h2.console.settings.trace": "true",
    "logging.level.root": "info",
    "spring.datasource.hikari.maximum-pool-size": "5",
    "spring.jpa.hibernate.ddl-auto": "update",
    "spring.jpa.show-sql": "true",
    "spring.datasource.hikari.minimum-idle": "10000"
}
}
```


This information is very useful for checking that an application is using the expected configuration. It's particularly useful when an application is deployed to a test or prod environment and using externalized configuration such as an external properties file or environment variables.