Spring Overview

The Spring framework is Java based framework for building web and enterprise applications. Spring at its core is a dependency injection container that provides flexibility to configure beans in multiple ways, such as XML, Annotations, and Java Config.

Over the years, the Spring framework grew exponentially by addressing the needs of modern business applications like security, support for NoSQL datastores, handling big data, batch processing, integration with other systems, etc. Spring, along with its sub-projects, became a feasible platform for building enterprise applications.

The Spring framework is very flexible and provides multiple ways of configuring the application components. with multiple configuration options, configuring Spring applications become complex and error-prone.
To address this complexity of configuration through its powerful **AutoConfiguration** mechanism Spring team created a utility framework I.e. Spring Boot.

Spring Boot is an opinionated framework following the "Convention Over Configuration" approach, to build the Spring-based applications quickly and easily. The main goal of Spring Boot is to quickly create Spring-based applications without requiring the developers to write the same boilerplate configuration again and again.

In recent years, the Microservices architecture has become the preferred architecture style for building complex enterprise applications. Spring Boot is a great choice for building Microservices-based applications using various Spring Cloud modules.

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.

What Is Spring Boot?

Spring Boot is an opinionated framework with the convention over configuration approach to build Spring-based applications quickly and easily. The main goal of Spring Boot is to quickly create Spring-based applications without requiring developers to write the same boilerplate configuration again and again.

The key Spring Boot features include:

- ✓ Spring Boot AutoConfiguration
- ✓ Spring Boot starters
- ✓ Spring Boot actuator
- ✓ Easy-to-use embedded servlet container support

Spring Boot AutoConfiguration

Spring Boot addresses the problem that , the traditional Spring applications requires complex configuration by eliminating the need to manually set up the boilerplate configuration.

Spring Boot uses an opinionated view of approach with Convention Over Configuration paradigm if **@EnableAutoConfiguration** or **@SpringBootApplication annotations are using** and configures various components automatically, by registering beans based on various criteria. The criteria can be:

- ✓ Availability of a particular class in a classpath
- ✓ Presence or absence of a Spring bean
- ✓ Presence of a system property
- ✓ Absence of a configuration file

For example, if we have the spring-webmvc dependency in our classpath, Spring Boot assumes we are trying to build a SpringMVC-based web application and automatically tries to register DispatcherServlet if it is not already registered.

If we have any embedded database drivers in the classpath, such as H2 or HSQL, and if we haven't configured a DataSource bean explicitly, then Spring Boot will automatically register a DataSource bean using in-memory database settings.

Spring Boot Starters

Spring Boot offers many starter modules with many of the commonly used technologies, like SpringMVC, JPA, MongoDB, Spring Batch, SpringSecurity, Solr, ElasticSearch, etc. These starters are pre-configured with the most commonly used library dependencies so we don't have to search for the compatible library versions and configure them manually.

For example, the **spring-boot-starter-data-jpa** starter module includes all the dependencies required to use Spring Data JPA, along with Hibernate library dependencies, as Hibernate is the most commonly used JPA implementation.

Spring Boot Actuator

getting the various details of an application running in production is crucial to many applications. The Spring Boot actuator provides a wide variety of such **production-ready features** without requiring developers to write much code. Some of the Spring actuator features are:

- ✓ **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

Embedded Servlet Container Support

Traditionally, while building web applications, we need to create WAR type modules and then deploy them on external servers like Tomcat, WildFly, etc. But by using Spring Boot, we can

create a JAR type module and embed the Servlet container in the application very easily so that the application will be a self-contained deployment unit. Also, during development, we can easily run the Spring Boot JAR type module as a Java application from the IDE or from the command-line using a build tool like Maven or Gradle.

The most important thing is that when we created spring boot application then we will get a simple Java class annotated with some magical annotation (@SpringApplication), which has a main() method. By running that main() method, we are able to run the application and access it at **http://localhost:8080/**. Where does the servlet container come from?

We added **spring-boot-starter-web**, which pulls **spring-boot-starter-tomcat** automatically. When we run the main() method, it starts tomcat as an embedded container so that we don't have to deploy our application on any externally installed tomcat server. What if we want to use a Jetty server instead of Tomcat? We simply exclude spring-boot-starter-tomcat from spring-boot-starter-web and include spring-boot-starter-jetty.

Creating A New Spring Boot Application

There are many ways of creating Spring Boot-based applications:

- ✓ 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**

Using the Spring Boot CLI

The Spring Boot CLI is a command-line tool to create and run the spring boot applications by perform the following steps:

1. Install the Spring Boot command-line tool by downloading the
spring-boot-cli-1.5.10.BUILD-20180130.154323-54-bin.zip

file from
https://repo.spring.io/snapshot/org/springframework/boot/spring-boot-cli/1.5.10.BUILD-SNAPSHOT/

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 1.5.10.BUILD-SNAPSHOT

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";
    }


}
```

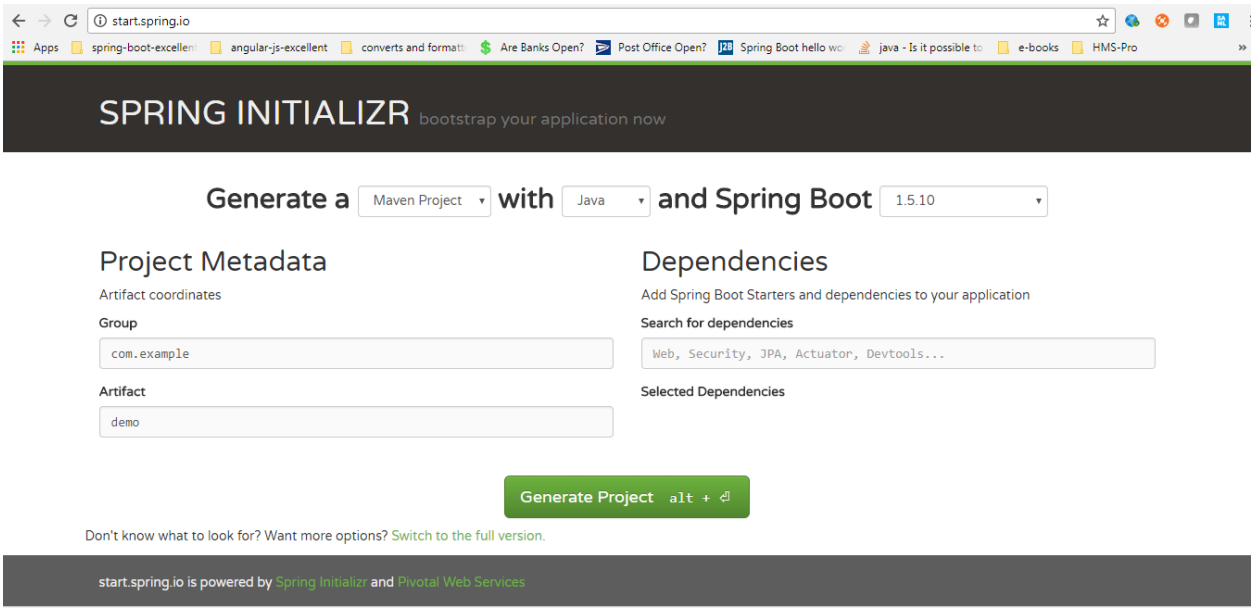
In order to run this Java application, go to the folder where LoginController.java is saved and execute the following command. The last few lines of the server start-up log will be similar to the following:

$spring run LoginController.java

```
2017-11-08 23:28:48.013  INFO 7080 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring FrameworkServlet 'dispatcherSe
rvlet'
2017-11-08 23:28:48.014  INFO 7080 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : FrameworkServlet 'dispatcherServlet': initializati
on started
2017-11-08 23:28:48.045  INFO 7080 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : FrameworkServlet 'dispatcherServlet': initializati
on completed in 31 ms
```

Using Spring Initializer

We can point our browser to **http://start.spring.io/** and see the project details, as shown in Figure:



1. Select Maven Project and Spring Boot version

2.Enter the Maven project details as follows:

•Group: com.nareshit

•Artifact: springboot-sample

•Name: springboot-sample

•Package Name: com.nareshit.sample

•Packaging: JAR

•Java version: 1.8

•Language: Java

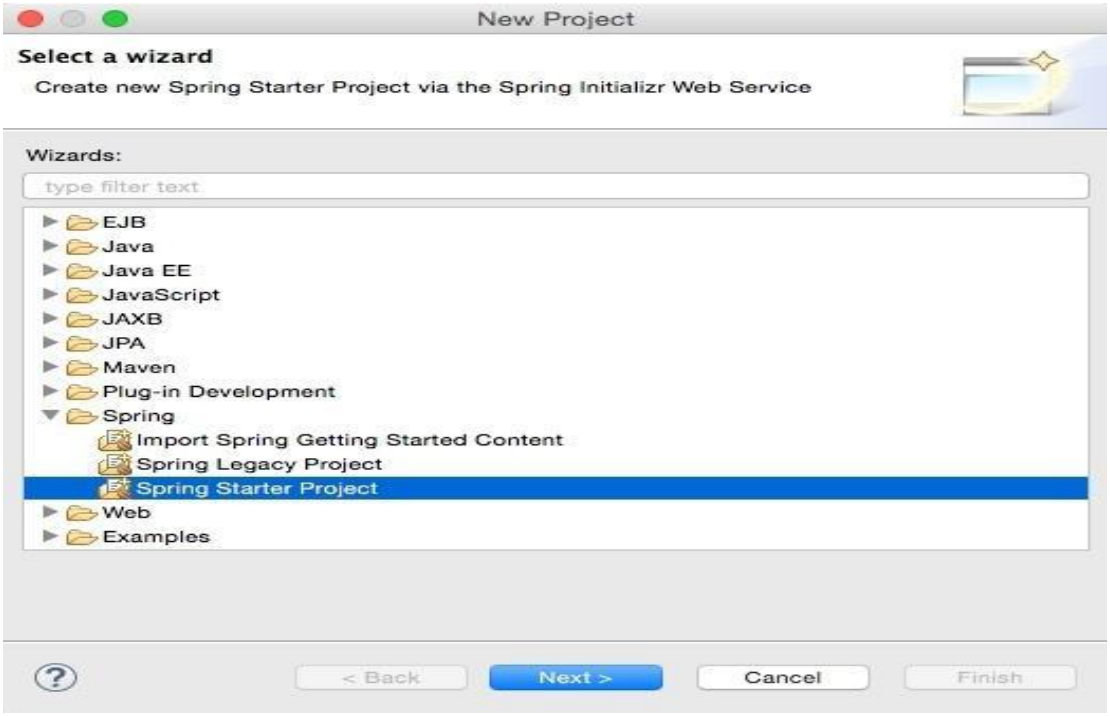
3. we can search for the starters if we are already familiar with their names or click on the Switch to the Full Version link to see all the available starters. We'll see many starter modules organized into various categories, like Core, Web, Data, etc. Select the Web check box from the Web category.

4. Click on the Generate Project button.

Now we can extract the downloaded ZIP file and import it into our favorite IDE.

Creating a new Spring Boot Project using the STS

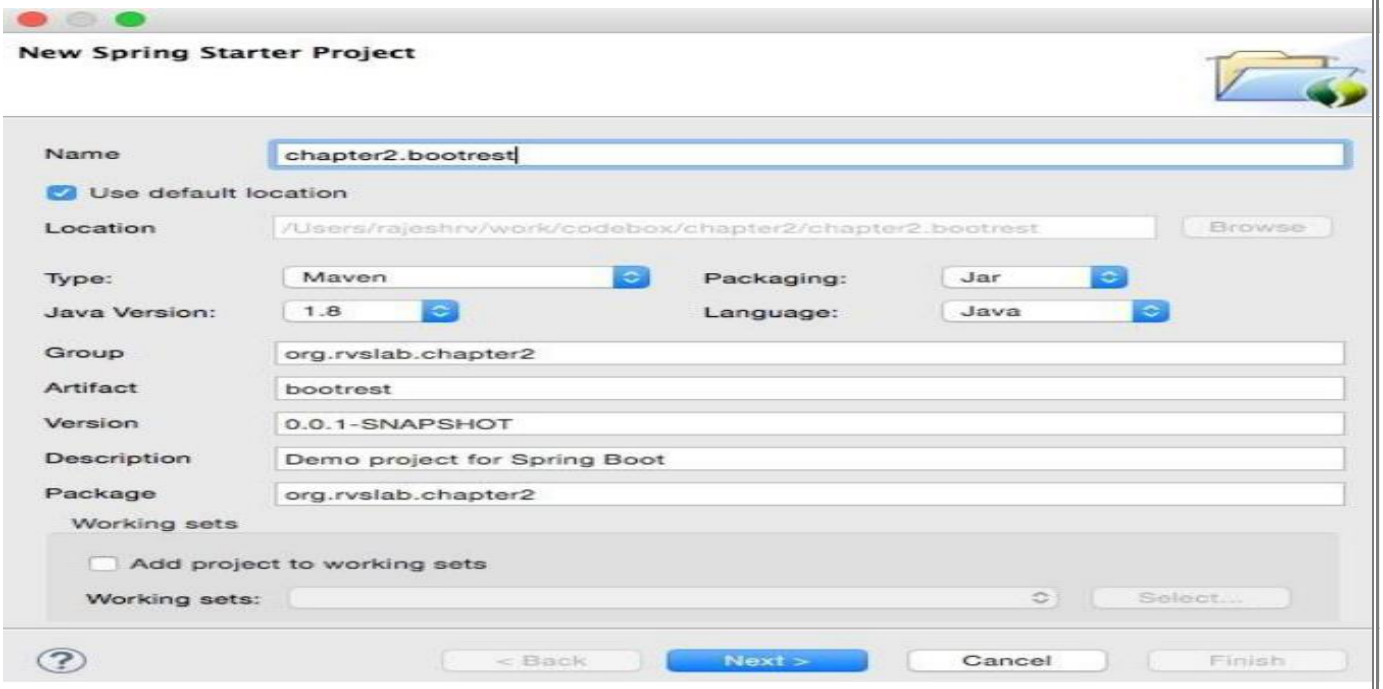
1. Open STS, right-click within the **Project Explorer** window, navigate to **New | Project**, and select **Spring Starter Project**, as shown in the following screenshot, and click on **Next**:



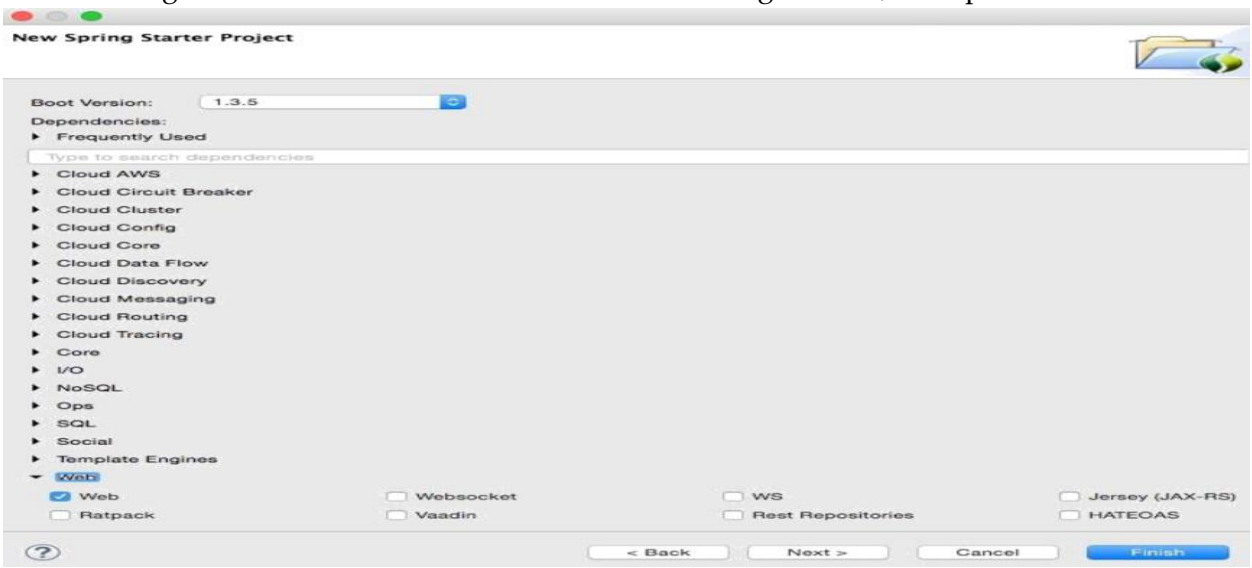
Spring Starter Project is a basic template wizard that provides a number of other starter libraries to select from.

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>

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.

Developing a Restful service – the legacy approach

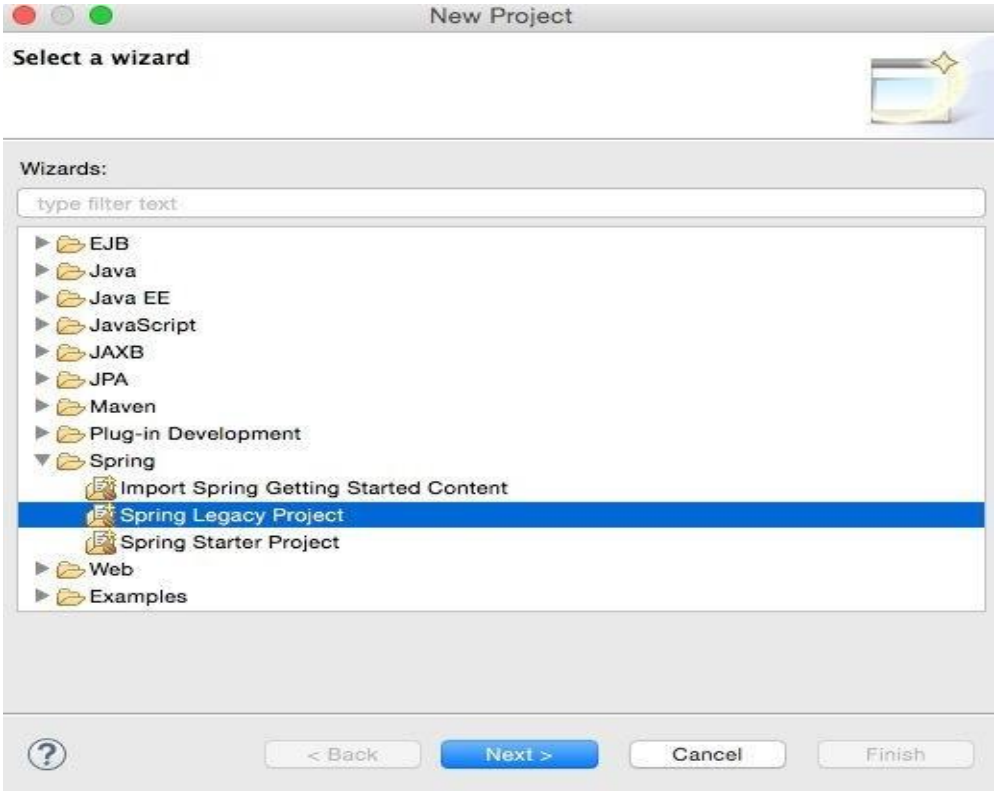
This example will review the traditional RESTful service development before jumping deep into Spring Boot. STS will be used to develop this REST/JSON service.

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

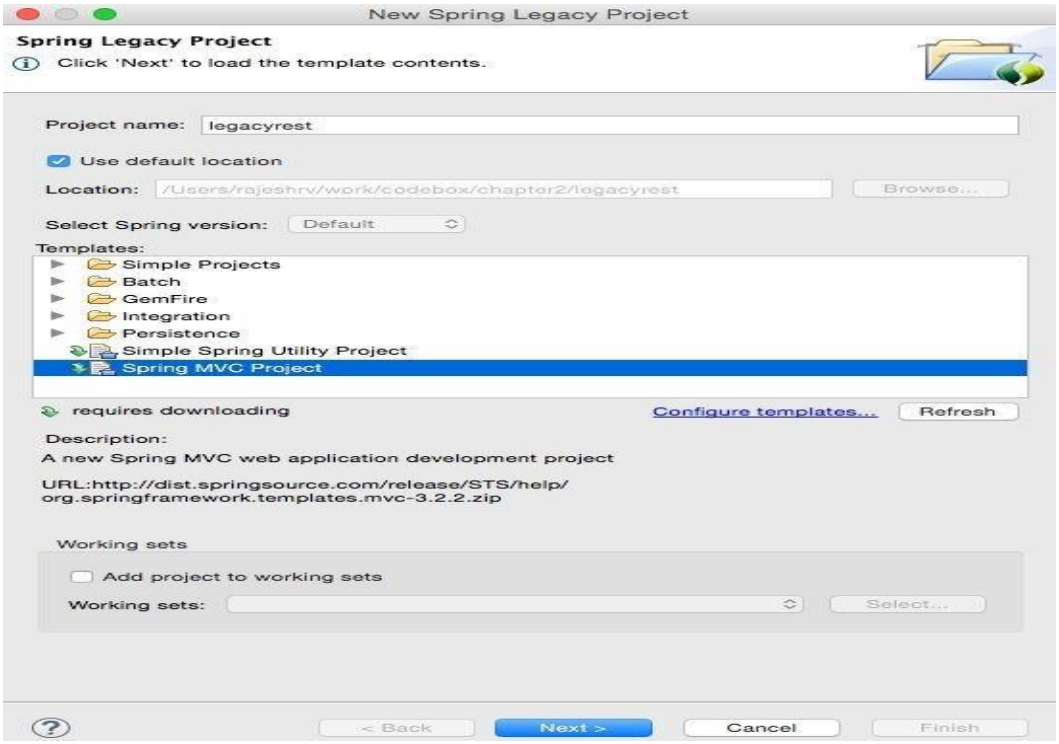
1. Start STS and set a workspace of choice for this project.

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

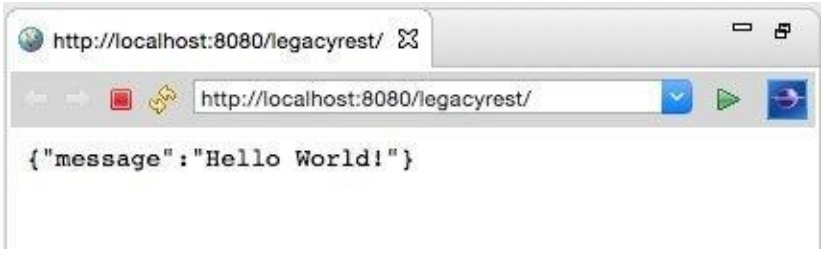
'appServlet': initialization completed in 906 ms

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 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 this is known as **Convention Over Configuration approach**.

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:

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.3.5.RELEASE)

2016-05-11 16:49:10.236  INFO 41130 --- [           main]
org.rvslab.chapter2.Application          : Starting Application on rvslab.local with
PID 41130 (/Users/rajeshrv/work/codebox/chapter2/chapter2.bootrest/target/classes
```

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.

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#string HttpMessageConverter 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)

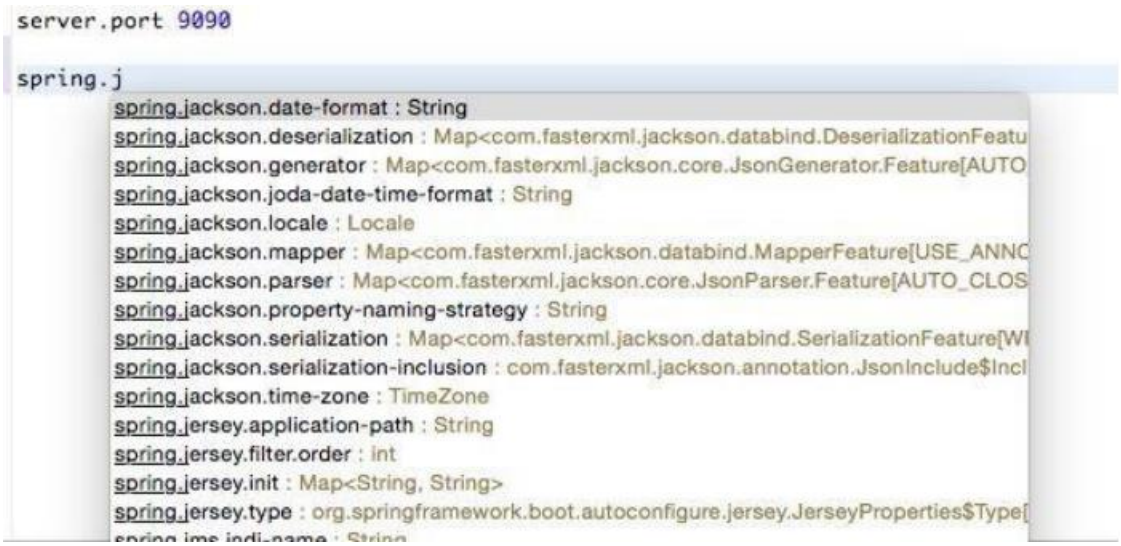
WebSocketAutoConfiguration.TomcatWebSocketConfiguration#websocketContainerCustomizer matched:
- @ConditionalOnJava (1.7 or newer) found 1.8 (OnJavaCondition)
- @ConditionalOnMissingBean (names: websocketContainerCustomizer; SearchStrategy:

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):

What is Profile?

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:

```
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:

```
<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:

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

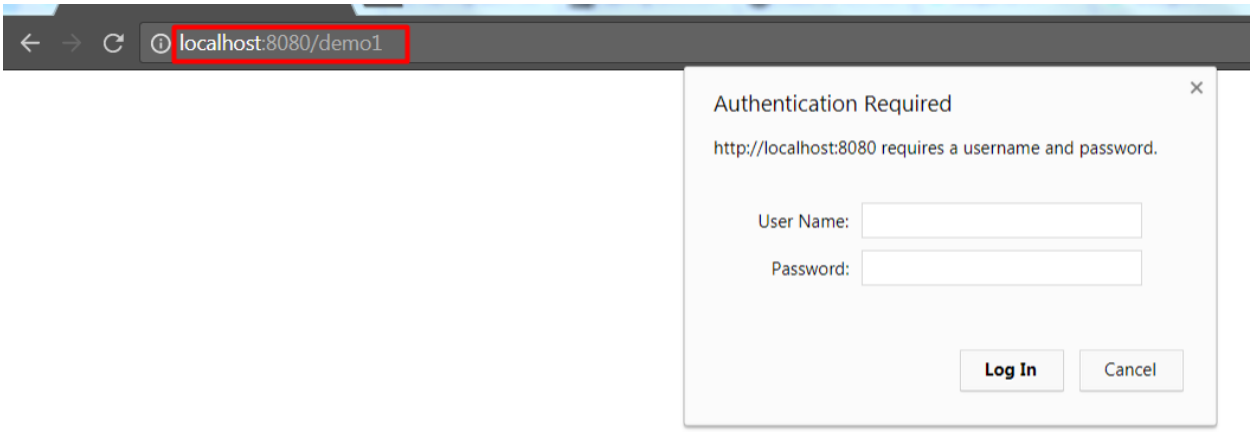
Open Application.java and add @EnableGlobalMethodSecurity to the Application class. This annotation will enable method-level security:

```
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:

```
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:

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

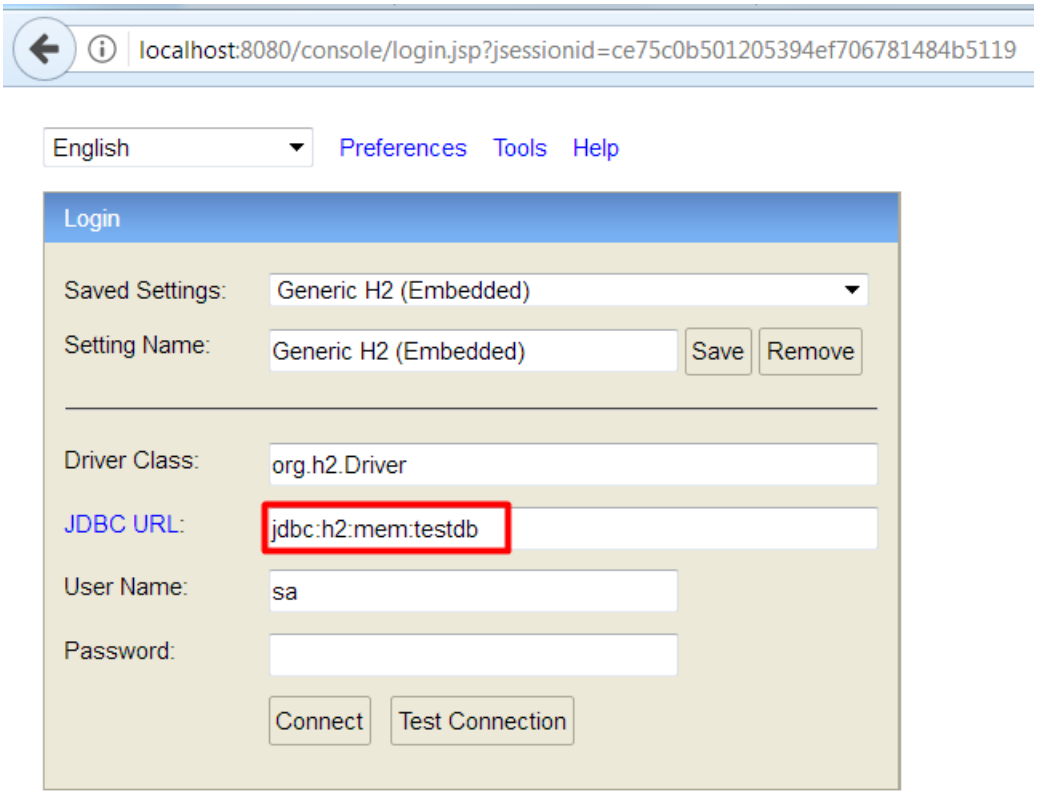
Now we need to tell spring boot which database we are going to use. The first thing we add is the database JDBC driver as a dependency to the pom. For the sake of simplicity we are going to use the **embedded in-memory database H2.**

```
<!-- 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:

```yaml
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
```

Step-4: Implements CommandLineRunner interface in DemoApplication.java

CommandLineRunner:

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)
```

Add Dynamic Query Methods

CrudRepository provided by the JPA doesn't have the methods to query the data based name. Now we'll add some finder methods to see how dynamic query generation based on method names works.

To get a user by name, use this:

User findByName(String name)

To search for users by name, use this:

List<User> findByNameLike(String name)

The preceding method generates a where clause like where u.name like ?1.

Suppose if we want to do a wildcard search, such as where u.name like %?1%. we can use @Query as follows:

@Query("select u from User u where u.name like %?1%")
List<User> searchByName(String name)

Suppose, if we wants to find the details by name, then use the following syntax:

@Query("select u from User u where u.name= :usernameSearch")
List<User> searchByName(@QueryParameter("usernameSearch")String name)

Using the Sort and Pagination Features

Suppose we want to get all users by their names in ascending order. We can use the findAll(Sort sort) method as follows:

```
Sort sort = new Sort(Direction.ASC, "name");
List<User> users = userRepository.findAll(sort);
```

We can also apply sorting on multiple properties, as follows:

```
Order order1 = new Order(Direction.ASC, "name");
Order order2 = new Order(Direction.DESC, "id");
Sort sort = Sort.by(order1, order2);
List<User> users = userRepository.findAll(sort);
```

The users will be ordered first by name in ascending order and then by ID in descending order.

Pagination

In many web applications, we'll want to show data in a page-by-page manner. Spring Data makes it very easy to load data in the pagination style. Suppose we want to load the first 25 users on one page. We can use Pageable and PageRequest to get results by page, as follows:

```
int size = 25;
int page = 0; //zero-based page index.
Pageable pageable = PageRequest.of(page, size);
Page<User> usersPage = userRepository.findAll(pageable);
```

The usersPage will contain the first 25 user records only. We can get additional details, like the total number of pages, the current page number, whether there is a next page, whether there is a previous page, and more.

- ✓ usersPage.getTotalElements();—Returns the total amount of elements.
- ✓ usersPage.getTotalPages();—Returns the total number of pages.
- ✓ usersPage.hasNext();
- ✓ usersPage.hasPrevious();
- ✓ List<User> usersList = usersPage.getContent();

We can also apply pagination along with sorting as follows:

```
Sort sort = new Sort(Direction.ASC, "name");
Pageable pageable = PageRequest.of(page, size, sort);
Page<User> usersPage = userRepository.findAll(pageable);
```

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.

Spring Boot, by default, pulls in **tomcat-jdbc-{version}.jar** and uses org.apache.tomcat.jdbc.pool DataSource to configure the DataSource bean.

Spring Boot checks the availability of the following classes and uses the first one that is available in the classpath.

- ✓ org.apache.tomcat.jdbc.pool.DataSource
- ✓ com.zaxxer.hikari.HikariDataSource
- ✓ org.apache.commons.dbcp.BasicDataSource
- ✓ org.apache.commons.dbcp2.BasicDataSource

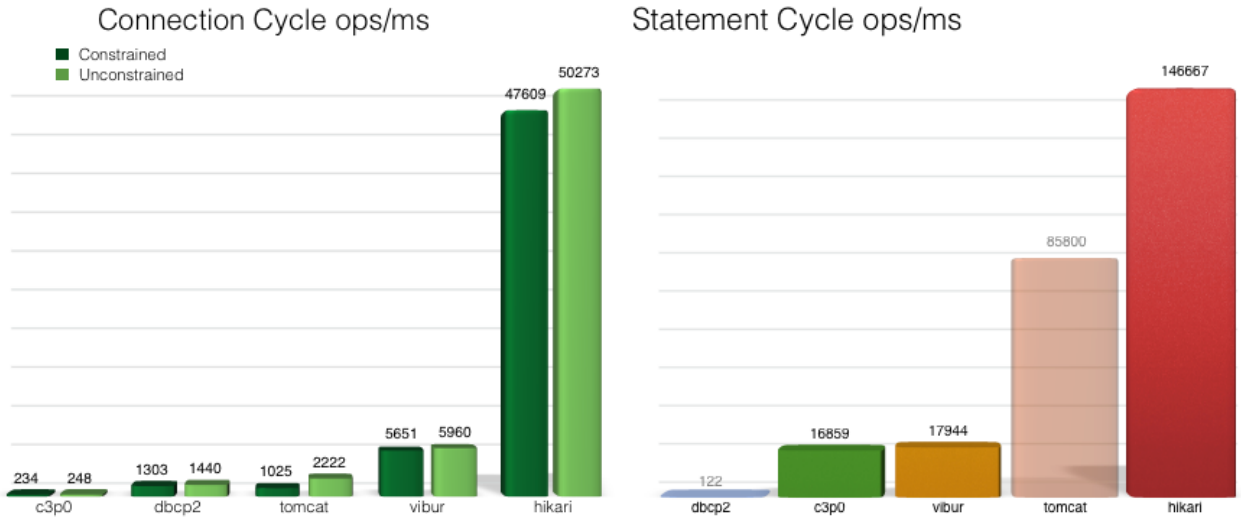
HikariCP Connection Pooling:

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:

```
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());
    }
}
```

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.

<u>This is how? Are we using hibernate so far?</u>

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:


```java
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

```
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.springframewc
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)
```

Note

Makes sure to configure the below property in properties file to work with transactions.

"spring.jpa.properties.hibernate.current_session_context_class=org.springframework.orm.hibernate4.SpringSessionContext"

Working with Multiple Databases

Spring Boot AutoConfiguration works out-of-the-box if we have single database to work with and provides plenty of customization options through its properties.But if our application demands more control over the application configuration, we can turn off specific auto configurations and configure the components by our self.

For example, we might want to use multiple databases in the same application. If we need to connect to multiple databases, we need to configure various Spring beans like DataSources, TransactionManagers, EntityManagerFactoryBeans, DataSourceInitializers, etc., explicitly.

Suppose we have an application where the security data has been stored in one database/schema and order-related data has been stored in another database/schema.
If we add the spring-boot-starter-data-jpa starter and just define the DataSource beans only, then Spring Boot will try to automatically create some beans (for example, TransactionManager), by assuming there will be only one data source. It will fail.
Now we'll see how we can work with multiple databases in Spring Boot and use the Spring Data JPA based application.

Step-1: Create a Spring Boot application with the data-jpa starter.
Configure the following dependencies in pom.xml:

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

<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
</dependency>
```

Step-2: Turn off the DataSource/JPA autoconfiguration. As we are going to configure the database related beans explicitly, we will turn off the DataSource/JPA autoconfiguration by excluding the AutoConfiguration classes, as

```java
package com.dotridge;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration;
import org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration;
import org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration;
import org.springframework.transaction.annotation.EnableTransactionManagement;

@SpringBootApplication(exclude = { DataSourceAutoConfiguration.class, HibernateJpaAutoConfiguration.class,
        DataSourceTransactionManagerAutoConfiguration.class })

@EnableTransactionManagement
public class SpringbootMultipleDSDemoApplication {

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

}
```

As we have turned off AutoConfigurations, we are enabling TransactionManagement explicitly by using the @EnableTransactionManagement annotation.

Step-3: Configure the datasource properties. Configure the Security and Orders database connection parameters in the application.properties file.

```
# Security Database Configurations
datasource.security.initialize=true
datasource.security.driver-class-name=com.mysql.jdbc.Driver
datasource.security.url=jdbc:mysql://localhost:3306/security
datasource.security.username=root
datasource.security.password=admin


# Oradres Database Configurations
datasource.orders.initialize=true
datasource.orders.driver-class-name=com.mysql.jdbc.Driver
datasource.orders.url=jdbc:mysql://localhost:3306/orders
datasource.orders.username=root
datasource.orders.password=admin

#Hibernate Properties
hibernate.hbm2ddl.auto=update
hibernate.show-sql=true
```

Here, we have used custom property keys to configure the two datasource properties.

Step-4: Create a security related JPA entity and a JPA repository. Then create a User entity, as follows:

```
@Entity
@Table(name="USERS")
public class User
{
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
private Integer id;

@Column(nullable=false)
private String name;

@Column(nullable=false, unique=true)
private String email;

private boolean disabled;
//setters & getters
}
```

Step-5: Create UserRepository as follows:

```
public interface UserRepository extends JpaRepository<User, Integer> {

}
```

Step-6:

```
@Entity
@Table(name="ORDERS")
public class Order{
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private Integer id;

    @Column(nullable=false, name="cust_name")
    private String customerName;

    @Column(nullable=false, name="cust_email")
    private String customerEmail;
```

```
    //setters & getters
}
```

Create the OrderRepository as follows:

```
public interface OrderRepository extends JpaRepository<Order, Integer> {
}
```

Step-7: Create the **SecurityDBConfig**.java configuration class. We will configure the Spring beans such as DataSource, TransactionManager, EntityManagerFactoryBean, and DataSourceInitializer by connecting to the Security database in SecurityDBConfig.java, as

Logging

Logging is a very important part of any application and it helps with debugging issues. Spring Boot, by default, includes **spring-boot-starter-logging** as a transitive dependency for the spring-boot-starter module. By default, Spring Boot includes SLF4J along with Logback implementations. Spring Boot has a LoggingSystem abstraction that automatically configures logging based on the logging configuration files available in the classpath.

If Logback is available, Spring Boot will choose it as the logging handler. We can easily configure logging levels within the **application.properties** file without having to create logging provider specific configuration files such as logback.xml or log4j.properties.

It is also set up to enable routing from common java logging libraries like java util Logging, Commons Logging, Log4j or SLF4J.

In the below 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.

```
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
```

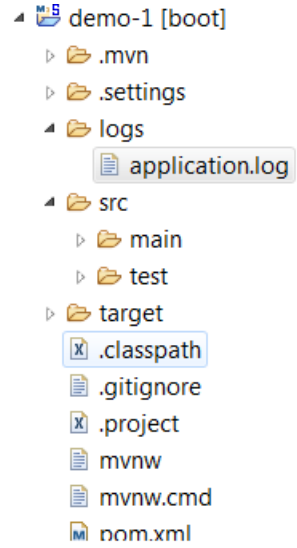
By default our logging message are printed on the console. This is not suitable for running the MicroServices later. So we are going to change it to the file based logging.

Logging to a File:

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

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

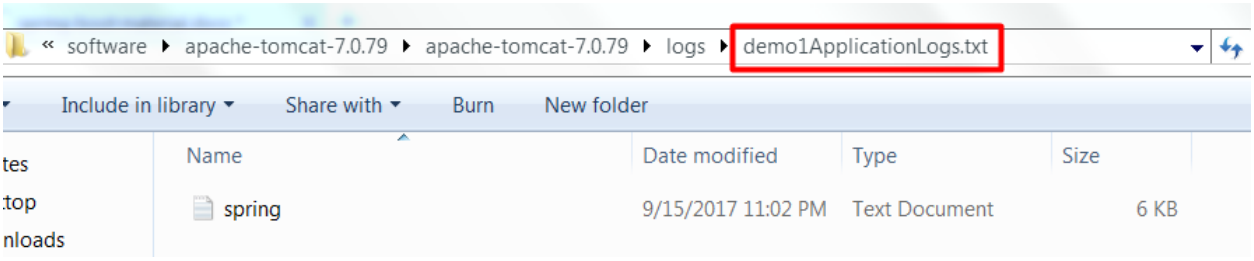
Now when we run the application, we could see a folder "logs" been created and inside in it application.log is created which holds all the application logs.



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**

<u>Changing the Log level</u>

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,
```

If we want to have more control over the logging configuration, create the logging provider specific configuration files in their default locations, which Spring Boot will automatically use. For example, if we place the **logback.xml** file in the root classpath, Spring Boot will automatically use it to configure the logging system.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>

    <!-- Console Appender -->
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n
            </pattern>
        </encoder>
    </appender>

    <!-- File appender -->
    <appender name="FILE" class="ch.qos.logback.core.FileAppender">
        <file>app.log</file>
        <encoder>
            <pattern>%date %level [%thread] %logger{10} [%file:%line] %msg%n
            </pattern>
        </encoder>
    </appender>

    <logger name="com.nareshit" level="DEBUG" additivity="false">
        <appender-ref ref="STDOUT" />
        <appender-ref ref="FILE" />
    </logger>

    <root level="INFO">
        <appender-ref ref="STDOUT" />
        <appender-ref ref="FILE" />
    </root>

</configuration>
```

Developer Tools

During development, we may need to change the code often and restart the server for those code changes to take effect. Spring Boot provides developer tools (the spring-boot-devtools module) that include support for quick application restarts whenever the application classpath content changes.

When we include the spring-boot-devtools module during development, the **caching of the view templates (Thymeleaf, Velocity, and Freemarkeretc) will be disabled automatically** so that we can see the changes immediately. We can see the list of configured properties at **org.springframework.boot. devtools.env.DevToolsPropertyDefaultsPostProcessor**.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
</dependency>
```

Note that in the production enabling the dev tools is not the suggested one and hence to ignore it in production specify spring-boot-devtools as optional by using <optional>true</optional> so that it won't be packaged in a fat JAR.

The Spring Boot developer tools trigger application restarts automatically whenever there is a change to the classpath content.

Whenever we change the class or properties files in the classpath, Spring Boot will automatically restart the server. We won't typically need to restart the server when static content, such as CSS, JS, and HTMLs, changes. So by default Spring Boot excludes these static resource locations from the file change watch list.

We can override this default exclusion list by configuring the spring.devtools.restart.exclude property.
spring.devtools.restart.exclude=assets/,resources/****

If we want to add locations to the restart exclude/include paths, use the following properties.
spring.devtools.restart.additional-exclude=assets/,setup-instructions/****
spring.devtools.restart.additional-paths=D:/global-overrides/

Spring Boot's restart mechanism helps increase developer productivity by automatically restarting the server after code changes. But at times, we may need to change multiple classes to implement some feature and it would be annoying if the server kept restarting after every file change. In this case, we can use the **spring.devtools.restart.trigger-file** property to configure a file path to watch for changes. That way, the server will restart only when the trigger-file has changed.

spring.devtools.restart.trigger-file=restart.txt

Note

Once we configure **spring.devtools.restart.trigger-file** and update the trigger file, the server will restart only if there are modifications to the files being watched. Otherwise, the server won't be restarted.

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:

ID	Description	Sensitive Default
info	Displays arbitrary application info.	false
health	Shows application basic health info for unauthenticated users and full details for authenticated users.	false
beans	Shows a list of all Spring beans configured in the application.	true
autoconfig	Displays an autoconfiguration report showing all autoconfiguration candidates and the reason they were/were not applied.	true
mappings	Displays a collated list of all @RequestMapping paths.	true
configprops	Displays a collated list of all @ConfigurationProperties.	true
metrics	Shows metrics information for the current application.	true
env	Exposes properties from Spring's ConfigurableEnvironment.	true
trace	Displays trace information (by default, the last 100 HTTP requests).	true
dump	Performs a thread dump.	true
loggers	Shows and modifies the configuration of loggers in the application.	true
auditevents	Exposes audit events information for the current application.	true
flyway	Shows any Flyway database migrations that have been applied.	true
liquibase	Shows any Liquibase database migrations that have been applied.	true
actuator	Provides a hypermedia-based "discovery page" for the other endpoints. Requires Spring HATEOAS to be on the classpath.	true
shutdown	Allows the application to be gracefully shut down (not enabled by default).	true

Additional Spring Boot Actuator Endpoints for SpringMVC Applications

ID	Description	Sensitive Default
docs	Displays documentation, including example requests and responses, for the Actuator's endpoints. Requires spring-boot-actuator-docs to be on the classpath.	false
heapdump	Returns a GZip-compressed hprof heap dump file.	true
jolokia	Exposes JMX beans over HTTP (when Jolokia is on the classpath).	true
logfile	Returns the contents of the logfile (if the logging.file or logging.path	true

properties have been set).

The sensitive actuator endpoints can be accessed by authenticated users only. for now we can disable security for actuator endpoints by setting the following property.
management.security.enabled=false

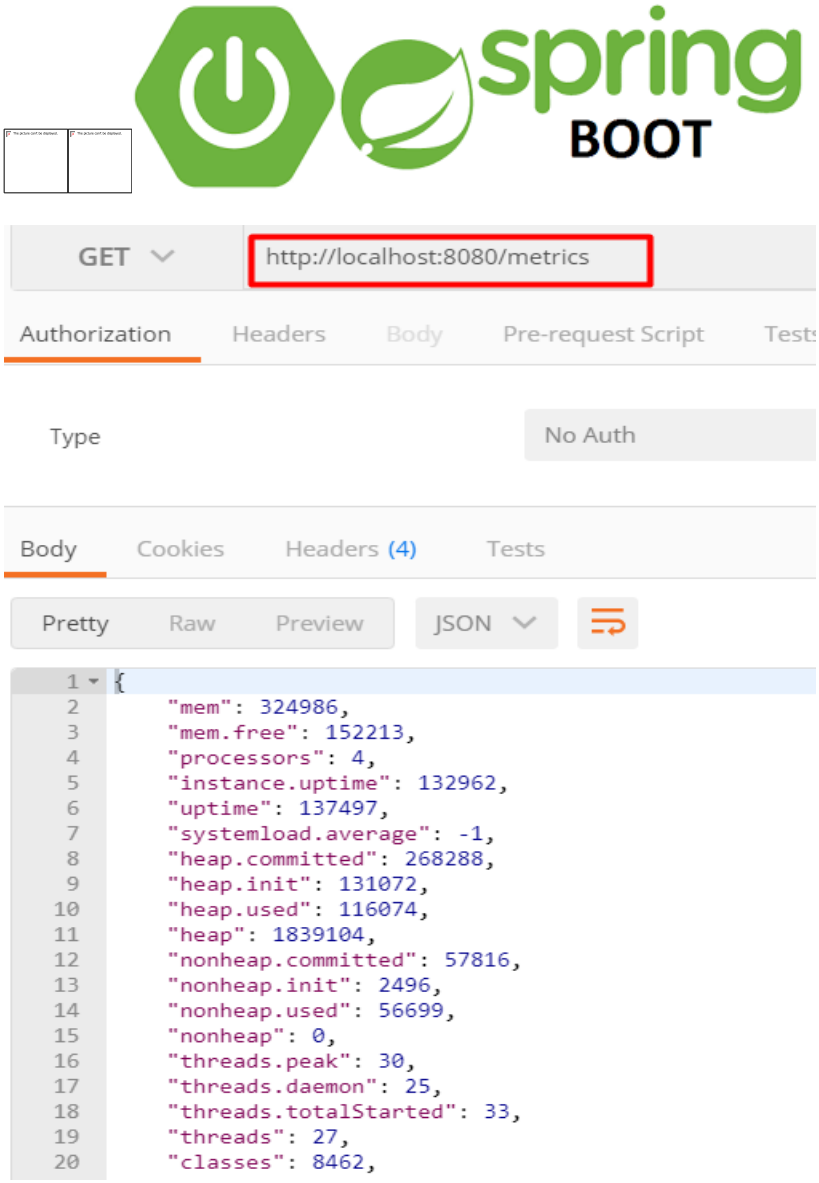
By default actuator endpoints run on the same HTTP port (server.port) with /application as the base path prefix.

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.

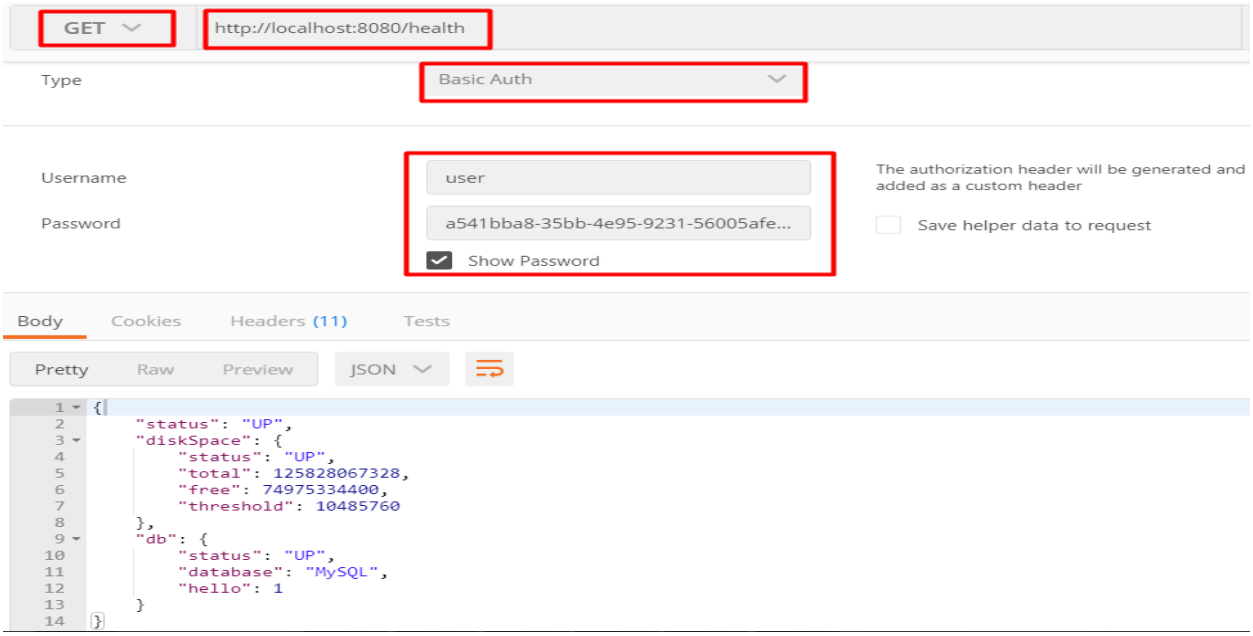
Health Checks

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.

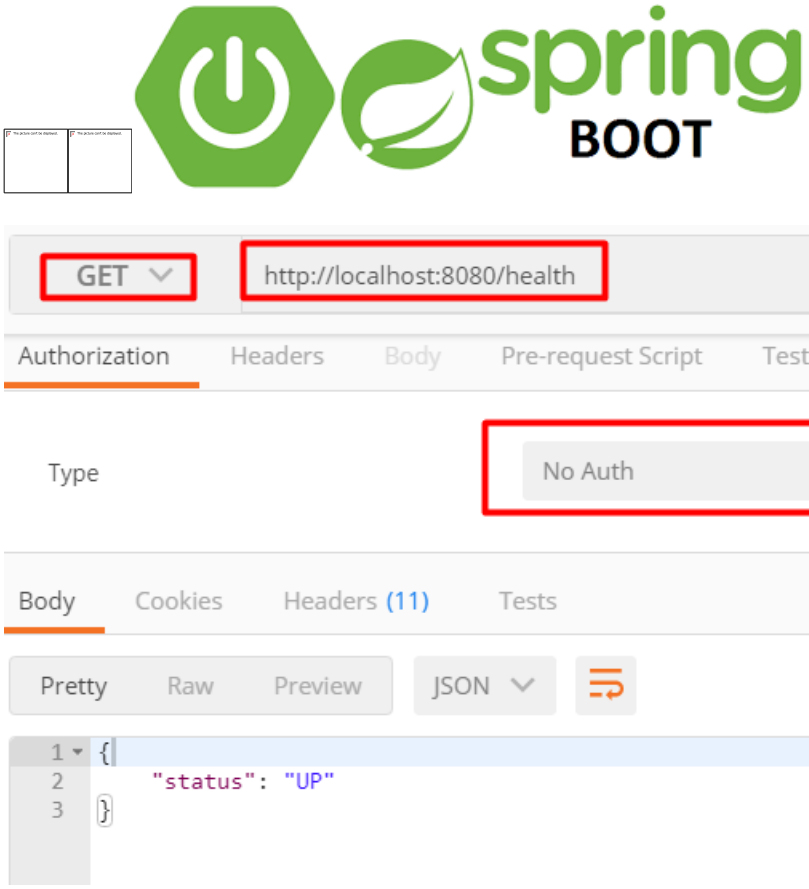
When the status UP is returned, a HTTP 200 response code is used. If the health check returns DOWN the response code is HTTP 503 (Service Unavailable). This is useful for load balancers or monitoring tools that may be calling the health check. They can derive the health status using the HTTP response 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

GET http://localhost:8080/info

Authorization Headers Body Pre-request Script Tests

Type No Auth

Body Cookies Headers (11) Tests

Pretty Raw Preview JSON

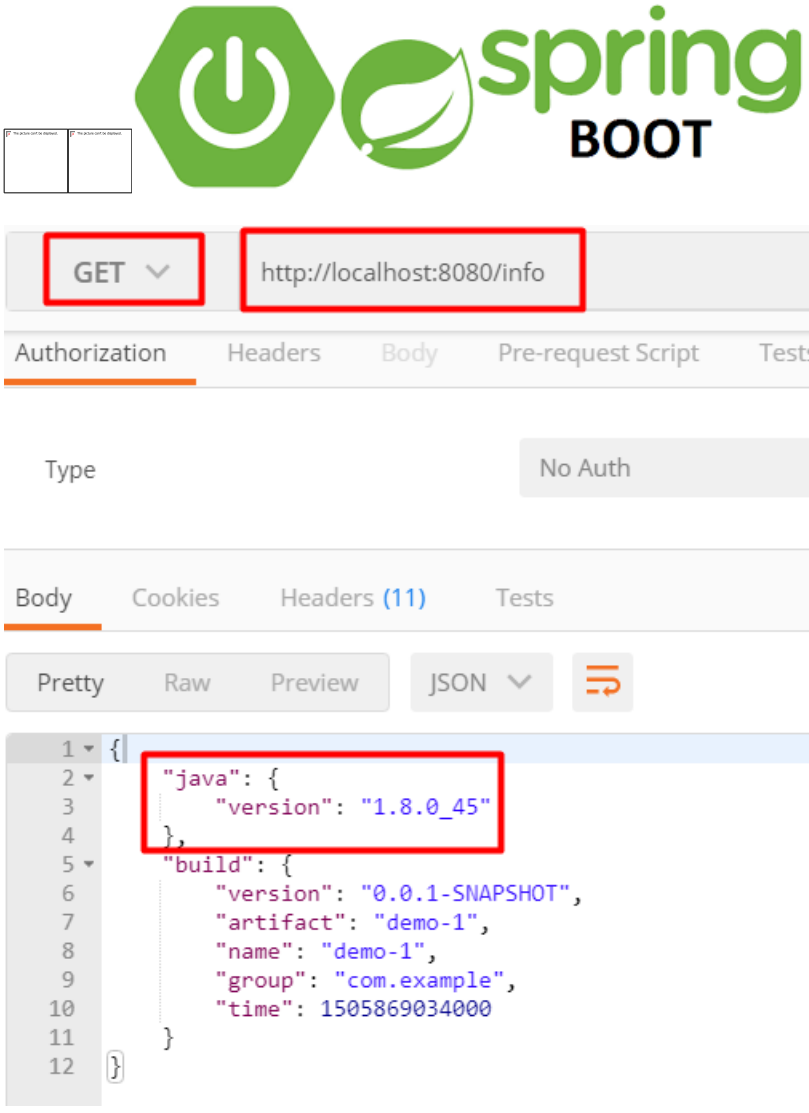
```
{
    "build": {
        "version": "0.0.1-SNAPSHOT",
        "artifact": "demo-1",
        "name": "demo-1",
        "group": "com.example",
        "time": 1505869034000
    }
}
```

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:

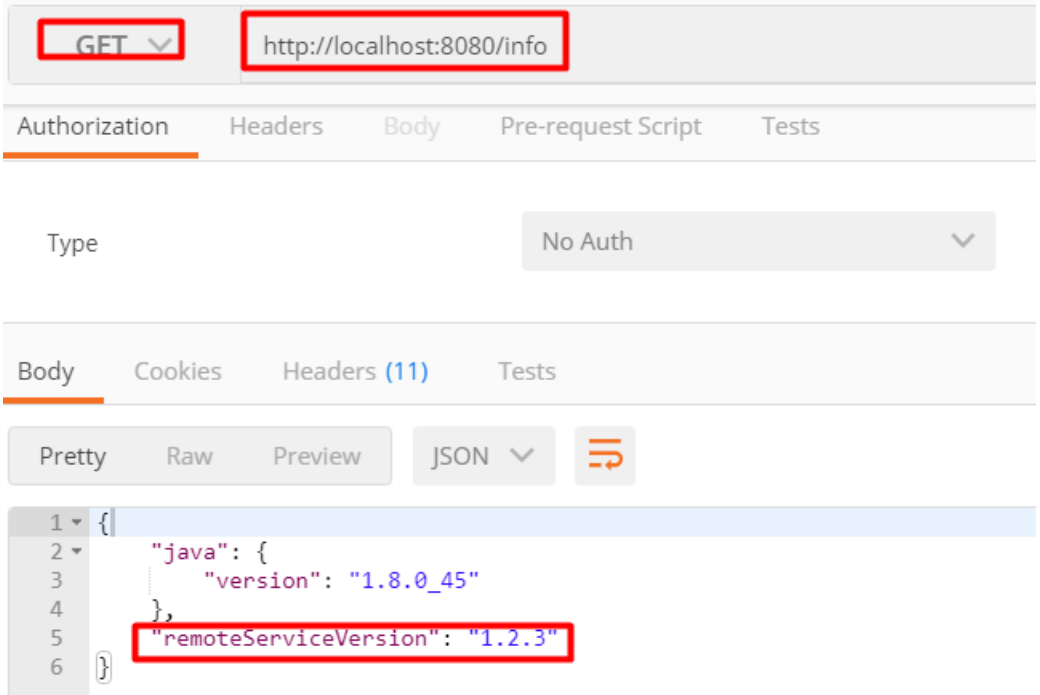


In the response above, *java.version* is populated using the configuration in *application.properties*.

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:

GET http://localhost:8080/metrics

Body Cookies Headers (4) Tests

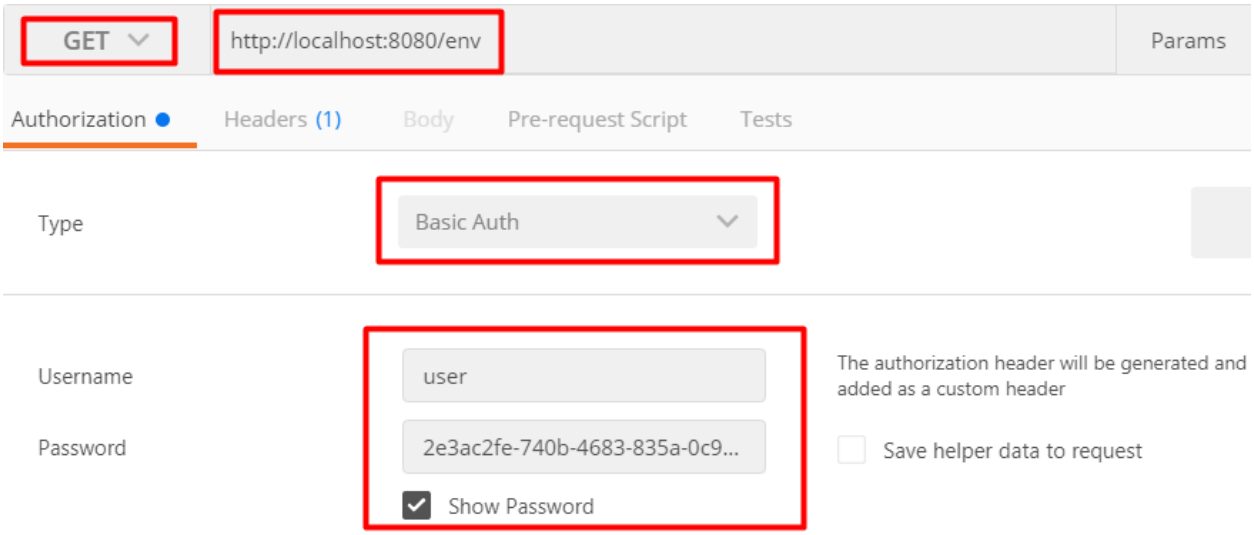
Pretty Raw Preview JSON


```
{
    "mem": 324986,
    "mem.free": 152213,
    "processors": 4,
    "instance.uptime": 132962,
    "uptime": 137497,
    "systemload.average": -1,
    "heap.committed": 268288,
    "heap.init": 131072,
    "heap.used": 116074,
    "heap": 1839104,
    "nonheap.committed": 57816,
    "nonheap.init": 2496,
    "nonheap.used": 56699,
    "nonheap": 0,
    "threads.peak": 30,
    "threads.daemon": 25,
    "threads.totalStarted": 33,
    "threads": 27,
    "classes": 8462,
    "classes.loaded": 8462,
    "classes.unloaded": 0,
    "gc.ps_scavenge.count": 9,
    "gc.ps_scavenge.time": 117,
    "gc.ps_marksweep.count": 2,
    "gc.ps_marksweep.time": 147,
    "httpsessions.max": -1,
    "httpsessions.active": 0,
    "datasource.primary.active": 0,
    "datasource.primary.usage": 0,
    "gauge.response.metrics": 68,
    "counter.status.200.metrics": 1
}
```


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.

The /shutdown Endpoint

The /shutdown endpoint can be used to gracefully shut down the application, which is not enabled by default. We can enable this endpoint by adding the following property to application.properties: **endpoints.shutdown.enabled=true**

After adding this property, you can send the HTTP POST method to http://localhost:8080/ application/shutdown to invoke the /shutdown endpoint.
Once the /shutdown endpoint is invoked successfully, we should see the following message:

```
{
    "message": "Shutting down, bye..."
}
```

Deploying Spring Boot Applications

Spring Boot supports embedded servlet containers, which makes deploying applications much easier, because we don't need an external application server setup. We can simply package our Spring Boot application as a JAR module and run it using the java -jar command.

However,we need to consider a few things while running applications in a production environment:
We can use profiles to externalize configuration properties per environment and run our application, activating desired profiles. But we don't want to specify sensitive data in configuration properties and commit it in the source code. Spring Boot provides various mechanisms to override the configuration properties when starting the application.

In recent years, *containerization* technologies like **Docker** have become a very popular way to run applications in both development and production environments. In complex applications, we may need to run multiple services like databases, search engines, log monitoring tools, etc. Installing all these services on the development environment is tedious. We can use Docker to spin up multiple containers with all the required services, without having to install all of them locally.

Running Spring Boot Applications in Production Mode

Spring Boot applications with JAR-type packaging are self-contained applications that can run easily and don't require any external application server setup. Once the application is packaged as a JAR, we can simply run the application as follows:

java -jar app.jar

If we have multiple profile configuration files, such as application-qa.properties and application-prod.properties, and can activate the desired profiles using the **spring.profiles.active** system property. Suppose we configured the datasource properties in the dev and prod profile configuration files, as

src/main/resources/application-dev.properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/doctor-service
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.jdbc.Driver

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

src/main/resources/application-prod.properties

```
spring.datasource.url=jdbc:h2:mem:patientdb
spring.datasource.username=sa
spring.datasource.password=
spring.datasource.driver-class-name=org.h2.Driver

spring.jpa.hibernate.ddl-auto=update

spring.h2.console.enabled=true
spring.h2.console.path=/h2Console
```

Now we can run the application in production by activating the prod profile, as follows:

java -jar -Dspring.profiles.active=prod app.jar

Suppose we have Spring components that should be activated only when we're running in a cloud environment. In that case, we can activate multiple profiles by specifying a comma-separated list of profile names.

java -jar -Dspring.profiles.active=prod,cloud app.jar

By activating prod and cloud profiles, all the default profile components (ie, components which are not associated to any specific environment) and components associated with prod and cloud profiles will be activated. But we don't want to configure sensitive information like actual production server credentials in these properties files and commit them to version control systems.

Spring Boot allows you to override the configuration parameters in various ways. We can override the properties using system properties as follows:

java -jar -Dserver.port=8585 app.jar

By specifying system properties, even if we configure the server.port property in the application-*. properties files, it will be overridden with server.port=8585 and the application will start on port 8585. Specifying a long list of properties like this can be cumbersome, so Spring Boot provides another mechanism to override the properties.

We can load the properties from multiple locations. SpringApplication will load properties from the application-*.properties files in the following locations:

- A /config subdirectory of the current directory
- The current directory
- A classpath /config package
- The classpath root

The properties defined in the top location take higher precedence over the properties defined in lower locations.

Suppose we have the app.jar in the /home/appserver/app1.0/ directory from which we run the application using the java -jar command.
Now we can create an application-prod.properties file in the /home/appserver/app1.0/config directory, which overrides the properties defined in the application-prod.properties file in classpath.
So we can configure all the default properties in the src/main/resources/application-*.properties files and override them using the config/application-*.properties file while running the application.

Note that we should not commit the config/applzication-*.properties files into version control systems, as they contain sensitive configuration details.

Deploying Spring Boot Application on Heroku

Heroku (https://www.heroku.com/) is a platform as a service (PaaS) that enables developers to build and run applications in the cloud. Heroku supports a wide variety of programming languages, including Ruby, Java, NodeJS, Python, PHP, Scala, and Go.

1) Create an account on Heroku and install the Heroku Toolbelt based on our operating system.

2) We can create an application using the heroku create command, which will automatically create a GIT remote (called heroku) and associate it with our local GIT repository.

3) Another option is to host our application code on a GitHub repository and link that repository to our Heroku application.

We will follow the second approach.

Step-1: Create a repository on GitHub.

I created a repository at https://github.com/Dotridge/cloudbatch

Step-2: Create a Spring boot application. Make sure spring boot application configured with the postgressql becase heroku is by default providing the postgressql in its production mode applications.
To configure postgressql add the following dependency in pom.xml file as:

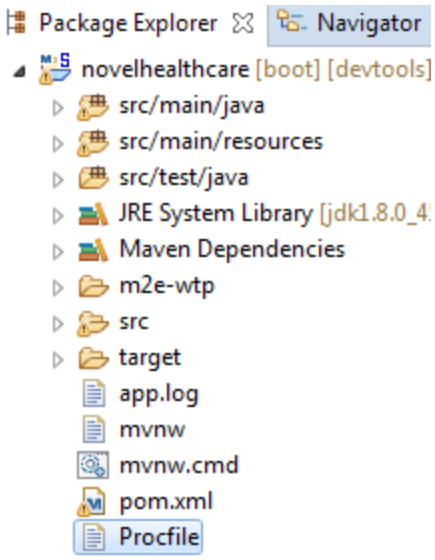
```
        <dependency>
            <groupId>org.postgresql</groupId>
            <artifactId>postgresql</artifactId>
        </dependency>
```

Step-3: Create application-heroku.properties file in src/main/resources and configure the postgressql properties as shown below:

```
spring.datasource.url=${JDBC_DATABASE_URL}
spring.datasource.driver-class-name=org.postgresql.Driver
spring.datasource.maxActive=10
spring.datasource.maxIdle=5
spring.datasource.minIdle=2
spring.datasource.initialSize=5
spring.datasource.removeAbandoned=true
spring.jpa.hibernate.ddl-auto=update
```

JDBC_DATABASE_URL is an environment variable that's generated dynamically by pointing to the Postgres database server. We can check the value of the **JDBC_DATABASE_URL** environment variable by running the following command:
heroku run echo \$JDBC_DATABASE_URL

Step-4: Create Procfile in Project Root Directory



And the content of the Procfile should be as :

```
web java -Dserver.port=$PORT -Dspring.profiles.active=heroku $JAVA_OPTS -jar target/novelhealthcare-0.0.1-SNAPSHOT.war
```

Note that the code configures the server.port property as a system property using -Dserver. port=$PORT, where the $PORT value is provided by Heroku dynamically. The code enables the heroku profile by specifying the -Dspring.profiles.active=heroku system property.

Now that we have the application code ready, we need to create an application on Heroku and link to the GitHub repository.

Go to **https://www.heroku.com/** and log in with our credentials. After a successful login, we will be redirected to the dashboard. On the Dashboard page, we can click on the New button on the top-right corner and select Create New App. There, we can provide the app name and select Runtime Selection. If we don't provide an app name, Heroku will generate a random name. Enter the application name and click on Create New App; it will take us to the application's Deploy configuration screen.

Click on the Resources tab and search for Postgres Add-on. Add the Heroku Postgres database. Click on the Deploy tab and, in the Deployment Method section, click on GitHub. Select our GitHub username, search for repository, and click Connect on the repository we want to link to.

Now we can click on Deploy Branch to deploy our application on Heroku. We can also enable automatic deployment of our application whenever we push changes to the GitHub repository by clicking on the Enable Automatic Deploys button.

Once the application is deployed, we can click on the Open App button in top-right corner, which will open our application home page in a new tab. If something goes wrong or we want to check the logs, choose More --> View Logs. We can also view logs from the terminal by running the following command.

```
heroku logs --tail --app <application_name>
```