/smlcodes /smlcodes /smlcodes

Spring Boot

TUTORIAL

Small Codes

Programming Simplified

A SmlCodes.Com Small presentation

In Association with Idleposts.com

For more tutorials & Articles visit SmlCodes.com

Spring Boot Tutorial

If you discover any errors on our website or in this tutorial, please notify us at support@smlcodes.com or smlcodes@gmail.com

First published on Sep 2017, Published by **SmlCodes.com**

Author Credits

Name	: **Satya Kaveti**
Email	: satyakaveti@gmail.com
Website	: smlcodes.com

Digital Partners

1. Introduction

Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run".

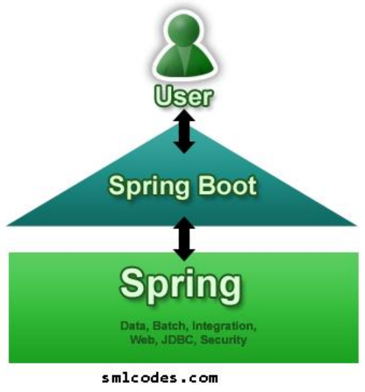
Spring vs SpringBoot

Spring: Spring started as a lightweight alternative to Java Enterprise Edition (J2EE). Spring offered a simpler approach to enterprise Java development, utilizing dependency injection and aspect-oriented programming to achieve the capabilities of EJB with plain old Java objects (POJOs).

But while spring was lightweight in terms of component code**, it was heavyweight in terms of configuration**. Initially, spring was configured with **XML & Spring 2.5** introduced **annotation-based component-scanning, even so, there was no escape from configuration.**

Spring boot: project is just a regular spring project that happens to leverage Spring Boot starters and auto-configuration. Spring Boot is not a framework, it is a way to ease to create **stand-alone application with minimal or zero configurations.**

Finally, Spring Boot is just spring. Spring projects would not have any XML configurations as part of it, everything will be handled by the project Spring Boot.



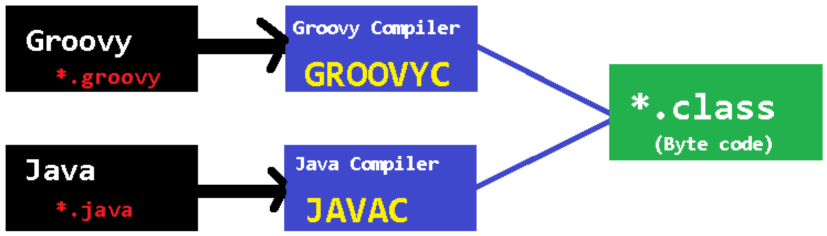
Spring Boot Features

- Create stand-alone Spring applications
- Embed **Tomcat, Jetty or Undertow directly** (no need to deploy WAR files)
- Provide opinionated **'starter' POMs to simplify your Maven configuration**
- **Automatically configure Spring whenever possible**
- Provide production-ready features such as metrics, health checks and externalized configuration
- Absolutely **no code generation and no requirement for XML configuration**

We can develop two flavors of Spring-Based Applications using Spring Boot

1. **Java-Based Applications**
2. **Groovy Application**

Groovy is also JVM language almost similar to Java Language. We can combine both Groovy and Java into one Project. Because like Java files, **Groovy files are finally compiled into *.class files** only. Both ***.groovy and *.java files are converted to *.class file (Same byte code format).**



Spring Boot Framework Programming model is inspired by Groovy Programming model. Spring Boot internally uses some Groovy based techniques and tools to provide default imports and configuration.

Creating Spring Boot Application

To create Spring Boot based applications The Spring Team (The Pivotal Team) has provided the following three approaches.

1. ***Using Maven***
2. ***Using Spring Initializer (http://start.spring.io/)***
3. ***Using Spring STS IDE***
4. ***Using Spring Boot CLI Tool***

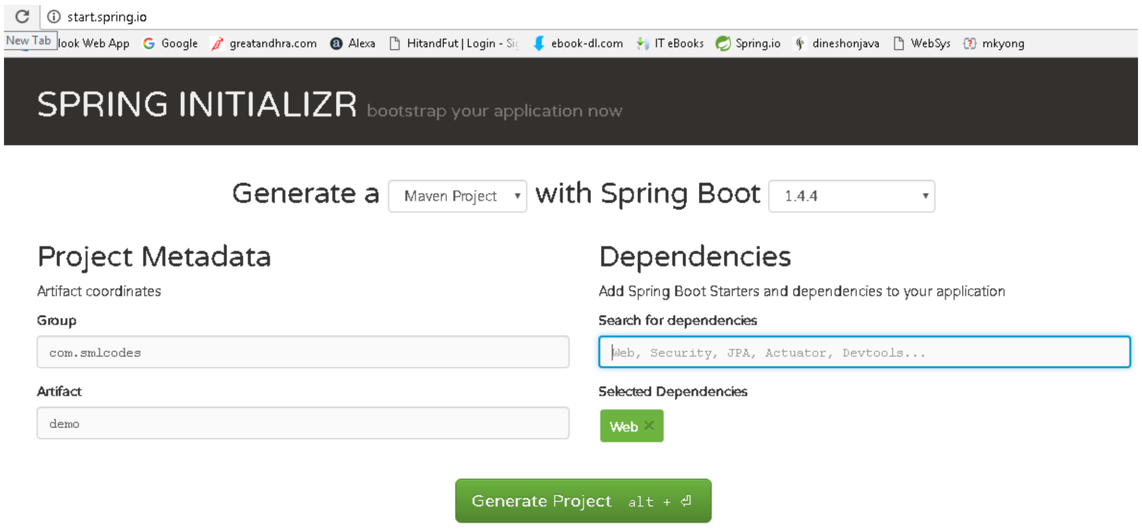
1. Spring Boot using Maven

Add maven dependencies in pom.xml & do maven build

```
<project>
        <parent>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-parent</artifactId>
                <version>1.5.6.RELEASE</version>
        </parent>

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

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

</project>
```

2.Spring Initializer

Spring Initializer provides an extensible API to **generate quick start projects**. It also provides a configurable service: you can see our default instance at https://start.spring.io. It provides a simple web UI to configure the project to generate and endpoints that you can use via plain HTTP.



Generate a Maven Project with Spring Boot 1.4.4

Project Metadata
Artifact coordinates
Group: com.smlcodes
Artifact: demo

Dependencies
Add Spring Boot Starters and dependencies to your application
Search for dependencies: Web, Security, JPA, Actuator, Devtools...
Selected Dependencies: Web

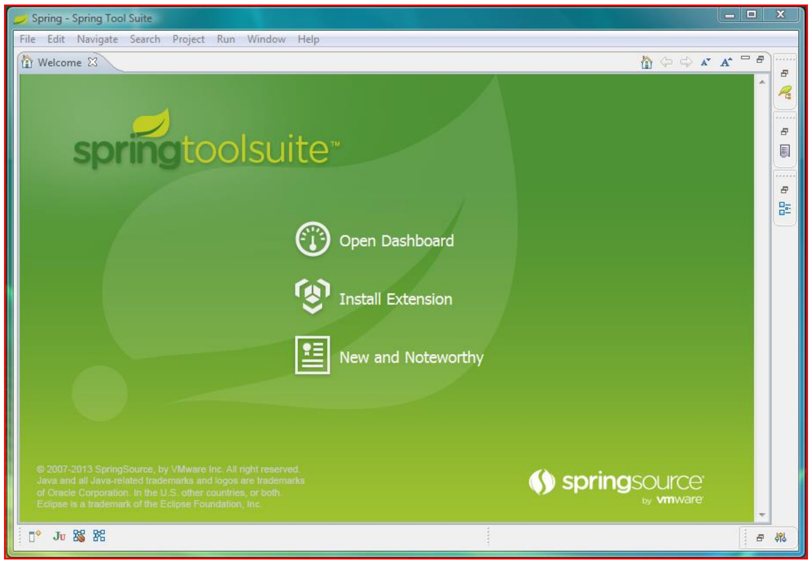
Generate Project alt + ⏎

It will generate the Artifact.zip file, extract it and run maven build: **`mvn clean install package`**

```
[INFO] --- spring-boot-maven-plugin:1.4.4.RELEASE:repackage (default) @ SpringBootDemo ---
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 01:27 min
[INFO] Finished at: 2017-01-30T18:55:21+05:30
[INFO] Final Memory: 27M/160M
[INFO] ------------------------------------------------------------------------
```

3.Spring STS IDE

The **Spring Tool Suite** is an Eclipse-based development environment that is customized for developing **spring** applications. We can download it from **here**.



4.Spring Boot CLI Tool

The Spring Boot CLI is a command line tool that can be used if you want to quickly prototype (creates project Structure) with Spring. It allows you to run Groovy scripts, which means that you have a familiar Java-like syntax, without so much boilerplate code.

You don't need to use the CLI to work with Spring Boot but it's definitely the quickest way to get a spring application off the ground.

You can download the Spring CLI distribution from the Spring software repository:

- spring-boot-cli-1.5.0.RELEASE-bin.zip
- spring-boot-cli-1.5.0.RELEASE-bin.tar.gz

SDKMAN! (The Software Development Kit Manager) can be used for managing multiple versions of various binary SDKs, including Groovy and the Spring Boot CLI. Get SDKMAN! from sdkman.io and install Spring Boot with

```
$ sdk install springboot
$ spring --version
Spring Boot v1.5.0.RELEASE
```

A simple web application that you can use to test your installation. Create a file called `app.groovy as`

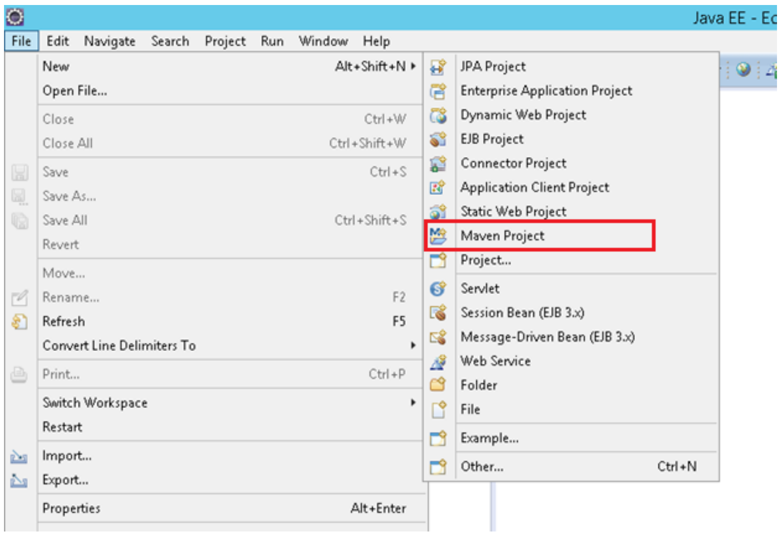
```
$ spring run app.groovy
```

It will take some time when you first run the application as dependencies are downloaded. Subsequent runs will be much quicker.

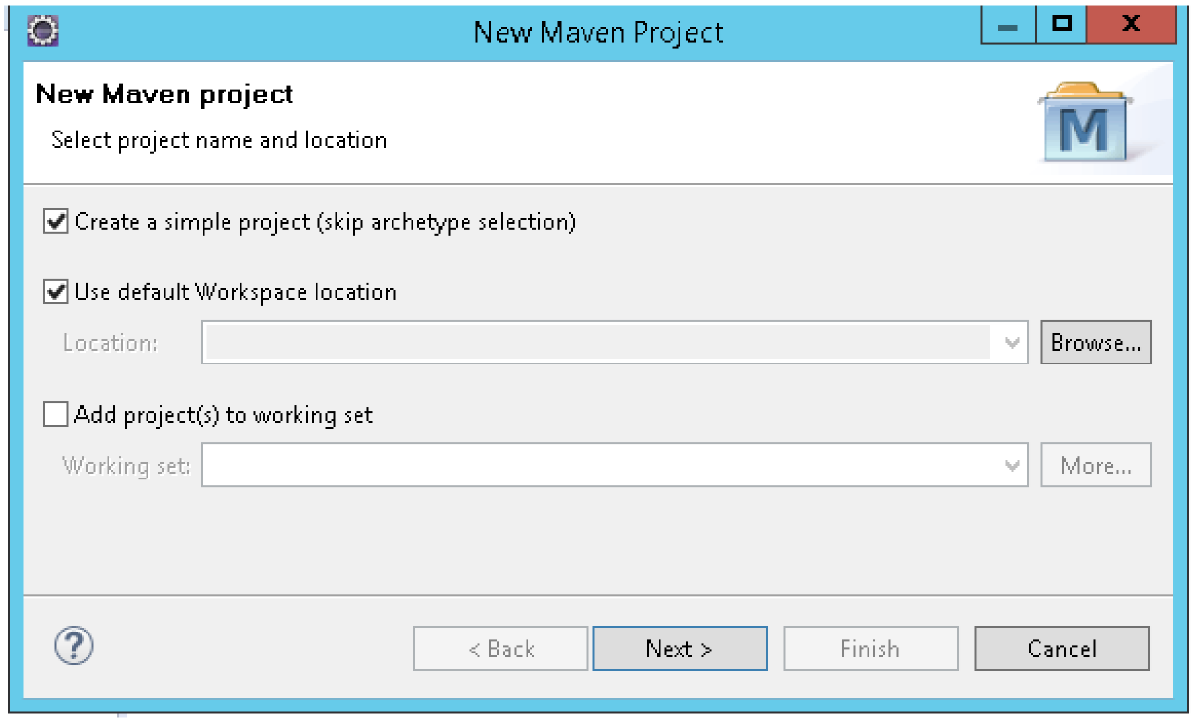
2. Spring Boot Examples

1.Spring Boot with maven and Eclipse Example

1. Open Eclipse > File > New > Maven Project



2.Tick 'Create a simple project (skip archetype selection) ' check box > click Next



New Maven Project

New Maven project

Select project name and location

☑ Create a simple project (skip archetype selection)

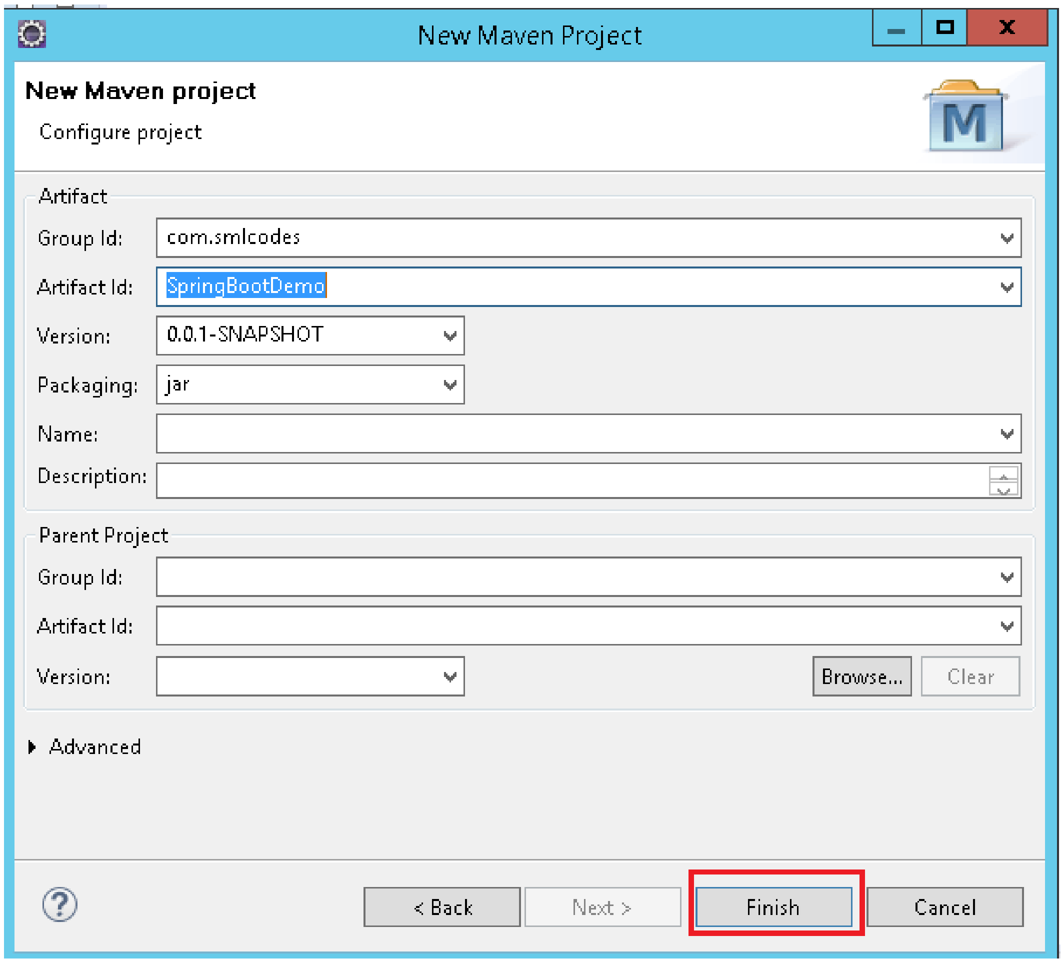
☑ Use default Workspace location

Location:

☐ Add project(s) to working set

Working set:

3. Provide Group Id (its your package), Artifact Id (project name) and click Finish



New Maven Project

New Maven project

Configure project

Artifact

Group Id: com.smlcodes

Artifact Id: SpringBootDemo

Version: 0.0.1-SNAPSHOT

Packaging: jar

Name:

Description:

Parent Project

Group Id:

Artifact Id:

Version:

Advanced

4. open pom.xml, add Spring Boot dependencies

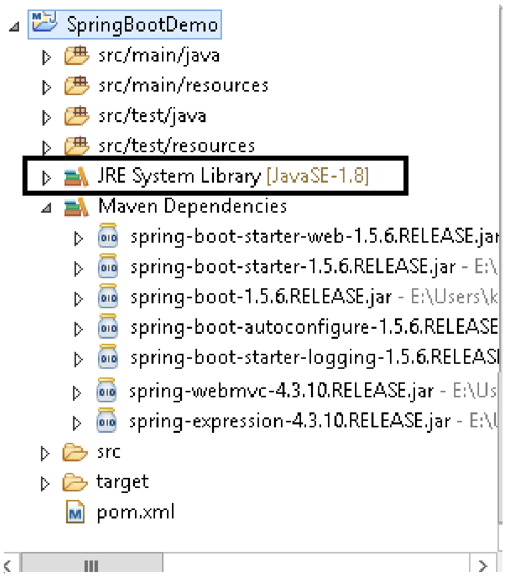
```
<project>
        <modelVersion>4.0.0</modelVersion>
        <groupId>com.smlcodes</groupId>
        <artifactId>SpringBootDemo</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <parent>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-parent</artifactId>
                <version>1.5.6.RELEASE</version>
        </parent>

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

        <properties>
                <java.version>1.8</java.version>
        </properties>
</project>
```

- `spring-boot-starter-parent`: is an existing project given by spring team which **contains Spring Boot supporting configuration data** (just configuration data, it won't download any jars), we have added this in a **<parent>** tag means, we are instructing Maven to consider our SpringBootHelloWorld project as a child to it

- `spring-boot-starter-web`: Starter for building web, including RESTful, applications using Spring MVC. Uses Tomcat as the default embedded container

5.Now right click on the application > Maven > Update Project,

if you observe the directory structure of the project, it will create a new folder named "*Maven Dependencies*" which contains all supporting. jars to run the Spring Boot application and the Java version also changed to **1.8.**


SpringBootDemo
src/main/java
src/main/resources
src/test/java
src/test/resources
JRE System Library [JavaSE-1.8]
Maven Dependencies
spring-boot-starter-web-1.5.6.RELEASE.jar
spring-boot-starter-1.5.6.RELEASE.jar - E:\
spring-boot-1.5.6.RELEASE.jar - E:\Users\k
spring-boot-autoconfigure-1.5.6.RELEASE
spring-boot-starter-logging-1.5.6.RELEASI
spring-webmvc-4.3.10.RELEASE.jar - E:\Us
spring-expression-4.3.10.RELEASE.jar - E:\
src
target
pom.xml

- If you observe pom.xml, we haven't included version number for **spring-boot-starter-web** but maven downloaded some jar files with some version(s) related to spring-boot-starter-web, that's because of Maven's parent child relation.
- While adding spring boot parent project, we included version as 1.5.6.RELEASE, so again we no need to add version numbers for the dependencies. As we know spring-boot-starter-parent contains configuration Meta data, this means, it knows which version of dependency need to be downloaded. So we no need to worry about dependencies versions. It will save lot of our time.

6.create a java class with main() method, in a pakage.com.smlcodes.app.SpringBootApp.java.

```java
package com.smlcodes.app;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringBootApp {
	public static void main(String[] args) {
		SpringApplication.run(SpringBootApp.class, args);
		System.out.println("****\n Hello, World \n ***");
	}
}
```

- **@SpringBootApplication** annotation, is the starting point for our Spring Boot application
- **SpringApplication.*run*(SpringBootApp.class, args);** it will bootstrapping the application

Remember, for every spring boot application we have to create a main class and that need to be annotate with @SpringBootApplication and bootstrap it

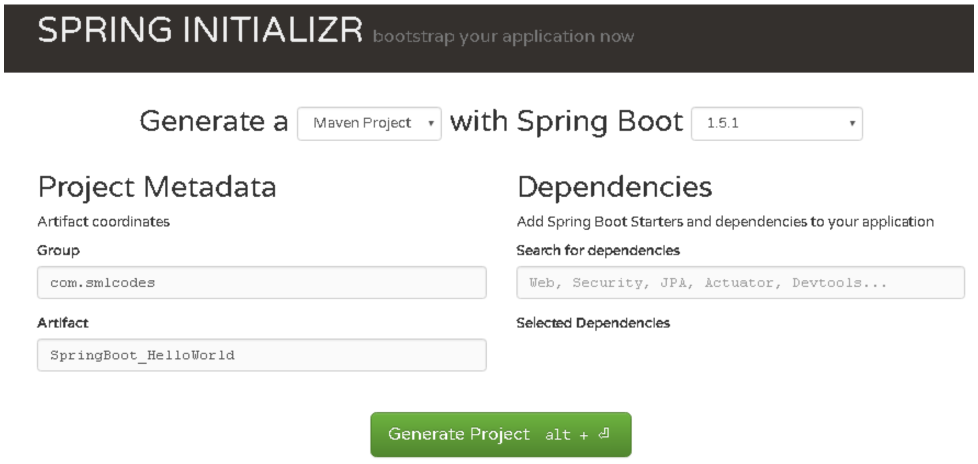
8.Finally, right click on the application > Run As > Java Application

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

2017-09-11 16:04:04.006  INFO 6724 --- [           main]
com.smlcodes.app.SpringBootApp           : Starting SpringBootApp on HYDPCMCSTS with PID 6724
2017-09-11 16:04:13.444  INFO 6724 --- [           main]
com.smlcodes.app.SpringBootApp           : Started SpringBootApp in 10.738 seconds (JVM
running for 12.252)
************
 Hello, World
 *****************
```

2.Spring Boot with Spring Initializr Example

We are using Spring Initializer (https://start.spring.io) to create a template for spring boot application

1. Go to https://start.spring.io , Choose Dependencies & Generate Project. Here we are not selecting any dependencies because it is just a Hello world program

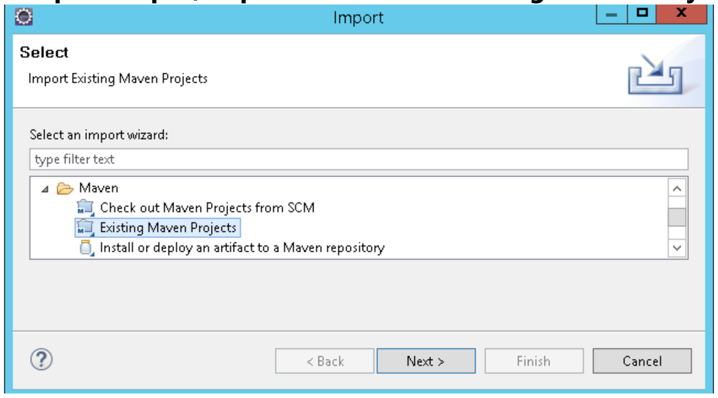

SPRING INITIALIZR bootstrap your application now

Generate a Maven Project with Spring Boot 1.5.1

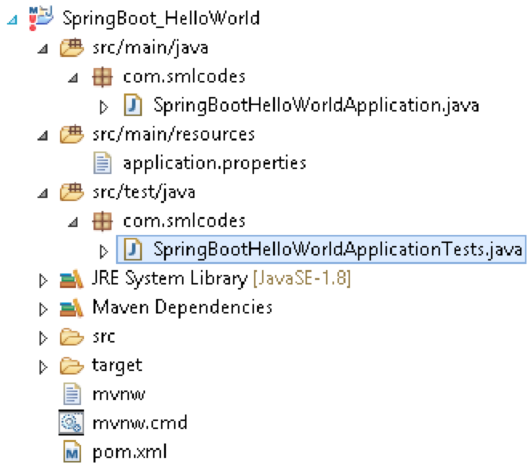
Project Metadata
Artifact coordinates
Group: com.smlcodes
Artifact: SpringBoot_HelloWorld

Dependencies
Add Spring Boot Starters and dependencies to your application
Search for dependencies: Web, Security, JPA, Actuator, Devtools...
Selected Dependencies

2. Open Eclipse, import→ Maven →Existing Maven Projects → Select Project →Finish


Import
Select
Import Existing Maven Projects
Select an import wizard:
type filter text
Maven
Check out Maven Projects from SCM
Existing Maven Projects
Install or deploy an artifact to a Maven repository

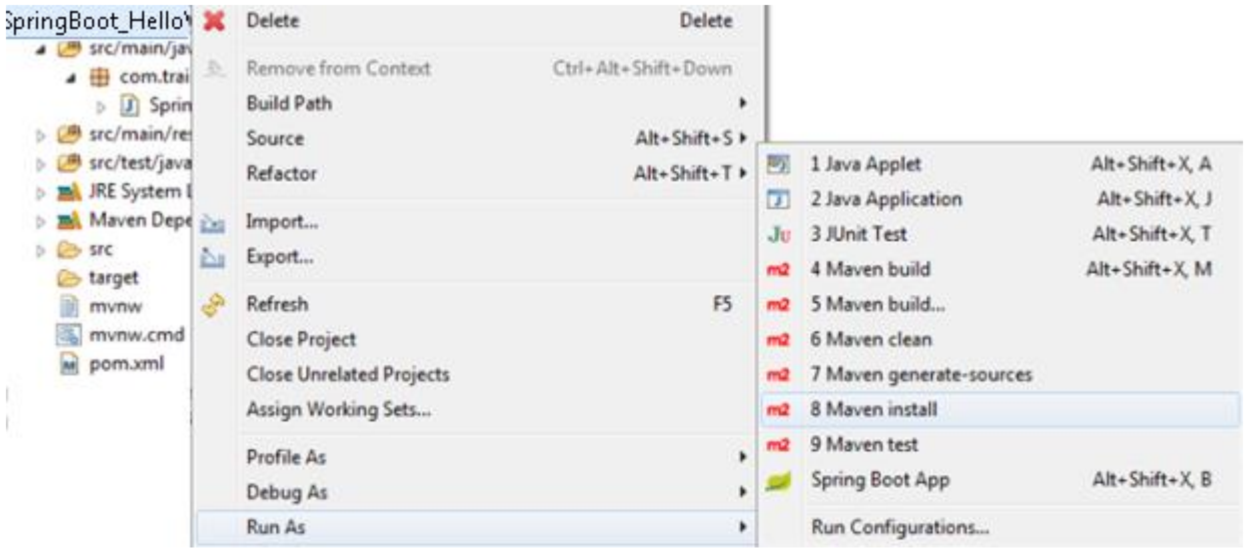
3. The Project structure will be as follows if we open eclipse Package explorer


```
SpringBoot_HelloWorld
  src/main/java
    com.smlcodes
      SpringBootHelloWorldApplication.java
  src/main/resources
    application.properties
  src/test/java
    com.smlcodes
      SpringBootHelloWorldApplicationTests.java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
  target
  mvnw
  mvnw.cmd
  pom.xml
```

4. If we open the pom.xml it contains only basic dependencies like `spring-boot-starter` which allows start spring boot application

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

5. Select Project, Right click Run as→ `maven install` to download and install the dependencies.



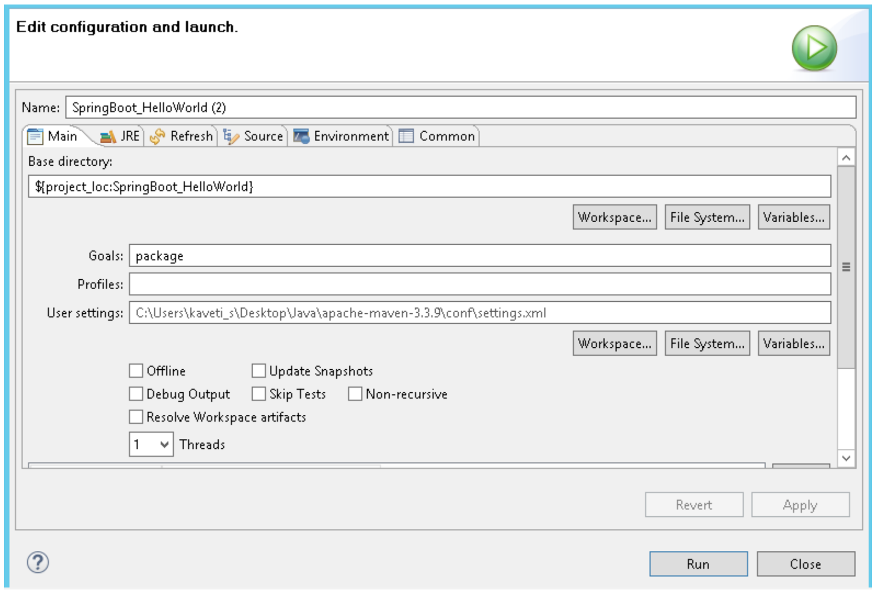
6. Spring boot generates the default java class which contains main() method. The main method calls the run method of SpringApplication. SpringApplication.run(SpringBootHelloWorldApplication.class, args); **This run method bootstraps the application starting spring which will run the embedded Tomcat Server. Let's add some helloworld message to print**

```java
package com.smlcodes;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringBootHelloWorldApplication {
	public static void main(String[] args) {
		SpringApplication.run(SpringBootHelloWorldApplication.class, args);
		System.out.println("============================================");
		System.out.println("Hello World, Spring Boot!!!!");
		System.out.println("============www.smlcodes.com==============");
	}
}
```

7. Select the Java file and right click RunAs → `Java Application`

```
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.5.1.RELEASE)
2017-01-31 11:05:32.745  INFO 29596 --- [           main]
s.c.a.AnnotationConfigApplicationContext : Refreshing
org.springframework.context.annotation.AnnotationConfigApplicationContext@646007f4: startup
date [Tue Jan 31 11:05:32 IST 2017]; root of context hierarchy
2017-01-31 11:05:43.108  INFO 29596 --- [           main]
o.s.j.e.a.AnnotationMBeanExporter        : Registering beans for JMX exposure on startup
2017-01-31 11:05:43.200  INFO 29596 --- [           main]
c.s.SpringBootHelloWorldApplication      : Started SpringBootHelloWorldApplication in 12.855
seconds (JVM running for 16.094)
=================================================
Hello World, Spring Boot!!!!
=============www.smlcodes.com===============
```

8. We can also run this application from the command line using the **jar file that is generated**. To get the jar file, select **pom.xml** right click **RunAs** → `Maven Build (2nd one)`, **goals=package, Apply & Run**


Edit configuration and launch.

Name: SpringBoot_HelloWorld (2)

Base directory: ${project_loc:SpringBoot_HelloWorld}

Goals: package

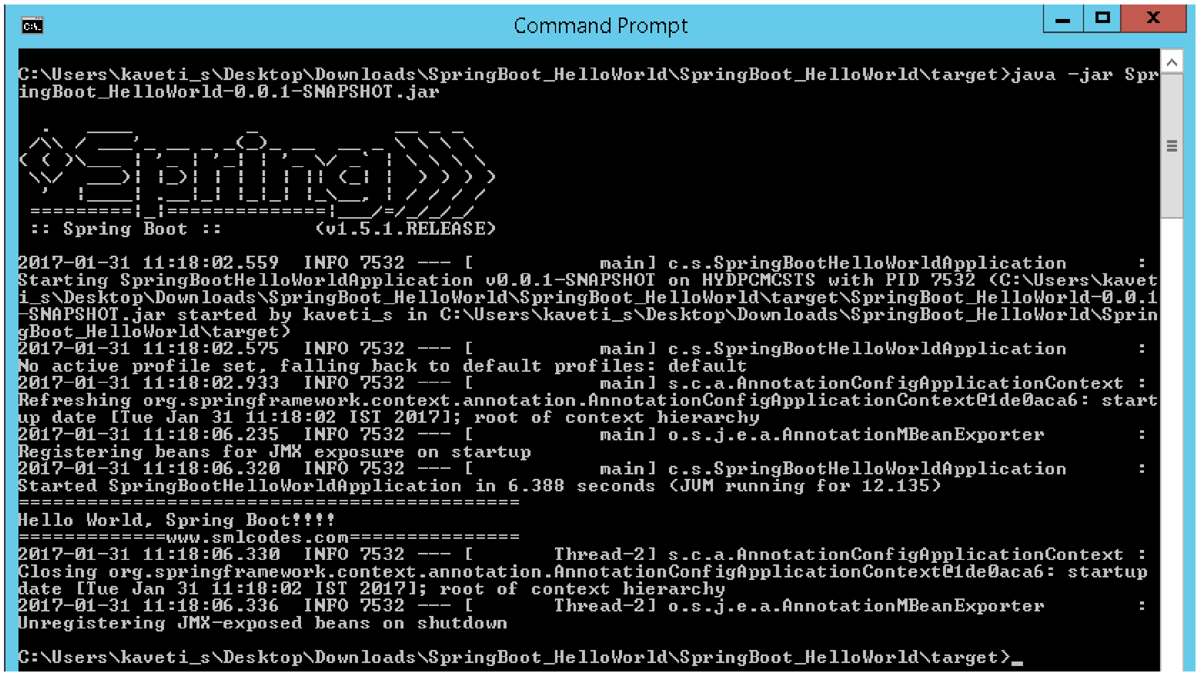
Profiles:

User settings: C:\Users\kaveti_s\Desktop\Java\apache-maven-3.3.9\conf\settings.xml

```
[INFO] Scanning for projects
[INFO] ------------------------------------------------------------------------
[INFO] Building SpringBoot_HelloWorld 0.0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
C:\Users\kaveti_s\Desktop\Downloads\SpringBoot_HelloWorld\SpringBoot_HelloWorld\target\SpringB
oot_HelloWorld-0.0.1-SNAPSHOT.jar
[INFO] --- spring-boot-maven-plugin:1.5.1.RELEASE:repackage (default) @ SpringBoot_HelloWorld
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

9. Open command line and go to the folder where your project is located. Next, move to the target folder and then type **java -jar <<filename>>.jar**.

```
java -jar SpringBoot_HelloWorld-0.0.1-SNAPSHOT.jar
```



```
C:\Users\kaveti_s\Desktop\Downloads\SpringBoot_HelloWorld\SpringBoot_HelloWorld\target>java -jar Spr
ingBoot_HelloWorld-0.0.1-SNAPSHOT.jar

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.5.1.RELEASE)

2017-01-31 11:18:02.559  INFO 7532 --- [           main] c.s.SpringBootHelloWorldApplication      :
Starting SpringBootHelloWorldApplication v0.0.1-SNAPSHOT on HYDPCMCSTS with PID 7532 (C:\Users\kavet
i_s\Desktop\Downloads\SpringBoot_HelloWorld\SpringBoot_HelloWorld\target\SpringBoot_HelloWorld-0.0.1
-SNAPSHOT.jar started by kaveti_s in C:\Users\kaveti_s\Desktop\Downloads\SpringBoot_HelloWorld\Sprin
gBoot_HelloWorld\target)
2017-01-31 11:18:02.575  INFO 7532 --- [           main] c.s.SpringBootHelloWorldApplication      :
No active profile set, falling back to default profiles: default
2017-01-31 11:18:02.933  INFO 7532 --- [           main] s.c.a.AnnotationConfigApplicationContext :
Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@1de0aca6: start
up date [Tue Jan 31 11:18:02 IST 2017]; root of context hierarchy
2017-01-31 11:18:06.235  INFO 7532 --- [           main] o.s.j.e.a.AnnotationMBeanExporter        :
Registering beans for JMX exposure on startup
2017-01-31 11:18:06.320  INFO 7532 --- [           main] c.s.SpringBootHelloWorldApplication      :
Started SpringBootHelloWorldApplication in 6.388 seconds (JVM running for 12.135)
============================================
Hello World, Spring Boot!!!!
=============www.smlcodes.com===============
2017-01-31 11:18:06.330  INFO 7532 --- [       Thread-2] s.c.a.AnnotationConfigApplicationContext :
Closing org.springframework.context.annotation.AnnotationConfigApplicationContext@1de0aca6: startup
date [Tue Jan 31 11:18:02 IST 2017]; root of context hierarchy
2017-01-31 11:18:06.336  INFO 7532 --- [       Thread-2] o.s.j.e.a.AnnotationMBeanExporter        :
Unregistering JMX-exposed beans on shutdown

C:\Users\kaveti_s\Desktop\Downloads\SpringBoot_HelloWorld\SpringBoot_HelloWorld\target>_
```

How it works internally

1. We place the all the required **Spring boot starters in pom.xml** which are requires for implementing Spring Boot application. On loading project, the **all required starter dependencies are added automatically to the project**

2. By running Spring Boot main class, at **@SpringBootApplication** line it will do the **Auto configuration** things, it will automatically add all required annotations to Java Class ByteCode.

3. On Executing main() method **SpringApplication.run()** used to bootstrap and launches Spring Boot application.

1. Spring Boot Starters

Spring boot Starters are the one-stop-shop for all the Spring and related technology that we need**. For example**, if you want to get started using **Spring and JPA for database access**, just include the **spring-boot-starter-data-jpa** dependency in your project, and you are good to go.

All **official** starters follow a similar naming pattern; **spring-boot-starter-*,** where ***** is a particular type of application. The following are the some of the application starters are provided by Spring Boot under the **org.springframework.boot** group

Name	Description
`spring-boot-starter-web-services`	Starter for using Spring Web Services
`spring-boot-starter-web`	Starter for building web, including RESTful, applications using Spring MVC. Uses Tomcat as the default embedded container
`spring-boot-starter-test`	Starter for testing Spring Boot applications with libraries including JUnit, Hamcrest and Mockito
`spring-boot-starter-jdbc`	Starter for using JDBC with the Tomcat JDBC connection pool
`spring-boot-starter-jersey`	Starter for building RESTful web applications using JAX-RS and Jersey. An alternative to `spring-boot-starter-web`
`spring-boot-starter-aop`	Aspect-oriented programming with Spring AOP and AspectJ
`spring-boot-starter-security`	Starter for using Spring Security
`spring-boot-starter-data-jpa`	Starter for using Spring Data JPA with Hibernate
`spring-boot-starter`	Core starter, including auto-configuration support, logging,YML

In above Example we used `spring-boot-starter & spring-boot-starter-test` Starters which are configure in pom.xml

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

2.@SpringBootApplication Annotation

This annotation marks the class as a spring bean, configures the application by adding all the jars based on the dependencies and also scans the other packages for spring beans.

Spring Boot developers always have their main class annotated with ***@Configuration, @EnableAutoConfiguration and @ComponentScan.***

1. **@Configuration** – Specifies this class as a spring bean
2. **@EnableAutoConfiguration** – This tells how you want to configure Spring, based on the jar dependencies that you have added. & also Enable / Disable auto configuration
3. **@ComponentScan** – is to scan other packages for spring beans.
4. **`@Import`** - used to import additional configuration classes
5. **`@ImportResource`** -annotation to load XML configuration files

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

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

Since these annotations are so frequently used together Spring Boot provides a convenient **@SpringBootApplication** as an alternative. The @SpringBootApplication annotation **is equivalent to using @Configuration, @EnableAutoConfiguration and @ComponentScan** with their default attributes.

@SpringBootApplication = @Configuration + @ComponentScan + @EnableAutoConfiration.

The original @SpringBootApplication annotation class is defined as below

```
package org.springframework.boot.autoconfigure; @Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Configuration
@EnableAutoConfiguration
@ComponentScan
public @interface SpringBootApplication {
       Class<?>[] exclude() default {};

       String[] excludeName()default{};

       @AliasFor(annotation = ComponentScan.class, attribute = "basePackages")
       String[] scanBasePackages() default {};

       @AliasFor(annotation = ComponentScan.class, attribute = "basePackageClasses")
       Class<?>[] scanBasePackageClasses()default{};
}
```

3 SpringApplication Class

SpringApplication class is used to bootstrap and launch a Spring application from a Java main method. By default, class will perform the following steps to bootstrap your application:

- Create an appropriate ApplicationContext instance (depending on your classpath)
- Register a CommandLinePropertySource to expose command line arguments as Spring properties
- Refresh the application context, loading all singleton beans
- Trigger any CommandLineRunner beans

In most circumstances the static **run(Object, String[])** method can be called directly from your main method to bootstrap your application:

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

4. Embaded Servlet containers

The following embedded servlet containers are supported out of the box.By default we will get Tomcat

Name	Servlet Version	Java Version
Tomcat 8	3.1	Java 7+
Tomcat 7	3.0	Java 6+
Jetty 9.3	3.1	Java 8+
Jetty 9.2	3.1	Java 7+
Jetty 8	3.0	Java 6+
Undertow 1.3	3.1	Java 7+

5. Spring Boot Profiles (@Profile Annotation)

Spring Profiles provide a way to segregate parts of your application configuration and make it only available in certain environments. Any **@Component or @Configuration can** be marked with **@Profile** to limit when it is loaded

```
@Configuration
@Profile("production")
public class ProductionConfiguration {

    // ...

}
```

In the normal Spring way, you can use a **spring.profiles.active** Environment property to specify which profiles are active. You can specify the property in any of the usual ways, for example you could include it in your **application.properties**:**spring.profiles.active=dev,hsqldb** or specify on the command line using the **switch --spring.profiles.active=dev,hsqldb.**

6. Spring Boot Actuator

Spring Boot provides actuator to monitor and manage our application. Actuator is a tool which has HTTP endpoints. **When application is pushed to production, you can choose to manage and monitor your application using HTTP endpoints.**

To get production-ready features, we should use spring-boot-actuator module. We can enable this feature by adding it to the **pom.xml** file.

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

3. Spring Boot –Configurations

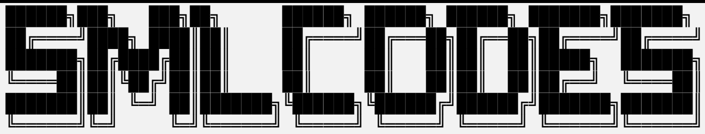
How to change Spring Boot Banner Text

The banner that is printed on startup can be changed by adding a **banner.txt** file to **src\main\resources** folder or your classpath, or by setting **banner.location** to the location of such a file.

You can also add a **banner.gif, banner.jpg or banner.png image file to your classpath**, or set a **banner.image.location property**. Images will be converted into an ASCII art representation and printed above any text banner.

```
//Default Banner
.   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.5.1.RELEASE)
2017-02-01 14:07:22.957  INFO 72716 --- [           main] c.s.SpringBootHelloWorldApplication
```

1. Go to any ANCII Text generator website & generate your logo. for ex: http://patorjk.com/
2. Create banner.txt under **Proj_Home\src\main\resources, paste the logo text**
3. **Refresh** the project & run Spring Boot Application.the banner will change as below



```
2017-02-01 14:07:22.957  INFO 72716 --- [           main]
c.s.SpringBootHelloWorldApplication
```

application.properties

Spring Boot provides a very neat way to load properties for an application. we can define properties in **application.properties** (`PROJ_HOME\src\main\resources\application.properties)` file the following way

```
db.name=smlcodesdb
db.username=smlcodes
db.password=wEB20R1XPJtg9
```

In traditional Spring application would have loaded up the properties in the following way

```
public class SmlcodesPropTest {
    @Value("${db.name}") //dbname is KEY here
    private String dbname;

    @Value("${db. username }")
    private String server_port;
```

In Spring boot it takes application.properties file to define a bean that can hold all the related properties in following way

```
@ConfigurationProperties(prefix = "db")
@Component
public class DBConfig {

	public String dbname;
	public String username;
	public String password;

	public String getDbname() {
		return dbname;
	}
	public void setDbname(String dbname) {
		this.dbname = dbname;
	}
	public String getUsername() {
		return username;
	}
	public void setUsername(String username) {
		this.username = username;
	}
	public String getPassword() {
		return password;
	}
	public void setPassword(String password) {
		this.password = password;
	}

}
```

@ConfigurationProperties is used to bind and validate some external Properties. If we want to validate before going to use, we have to place **@Validated** & place type of validation on the filed

```
@ConfigurationProperties(prefix="foo")
@Validated
public class FooProperties {

    @NotNull
    private InetAddress remoteAddress;

    // ... getters and setters

}
```

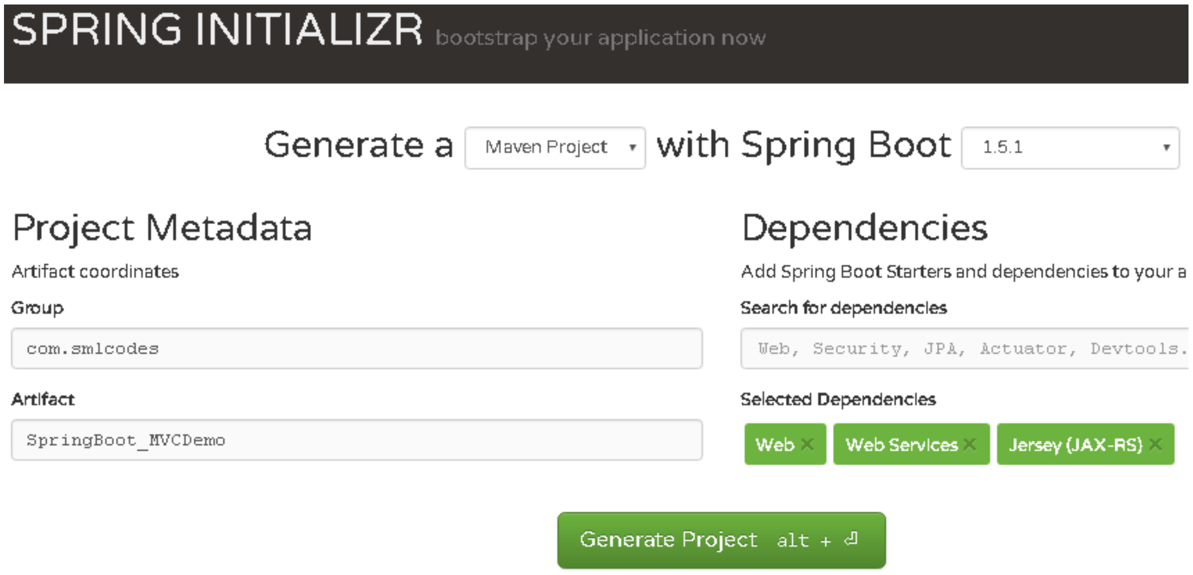
4. Spring Boot –MVC

In old Spring MVC lets you create special **@Controller** or **@RestController** beans to handle incoming HTTP requests. Methods in your controller are mapped to HTTP using **@RequestMapping** annotations.

SpringMVC example **@RestController** to serve JSON data

```
@RestController
@RequestMapping(value="/users")
public class SmlCodesRestController {

    @RequestMapping(value="/{user}", method=RequestMethod.GET)
    public User getUser(@PathVariable Long user) {
        // ...
    }

    @RequestMapping(value="/{user}/customers", method=RequestMethod.GET)
    List<Customer> getUserCustomers(@PathVariable Long user) {
        // ...
    }

    @RequestMapping(value="/{user}", method=RequestMethod.DELETE)
    public User deleteUser(@PathVariable Long user) {
        // ...
    }

}
```

Spring Boot MVC Example



SPRING INITIALIZR bootstrap your application now

Generate a Maven Project with Spring Boot 1.5.1

Project Metadata

Artifact coordinates

Group: com.smlcodes

Artifact: SpringBoot_MVCDemo

Dependencies

Add Spring Boot Starters and dependencies to your a

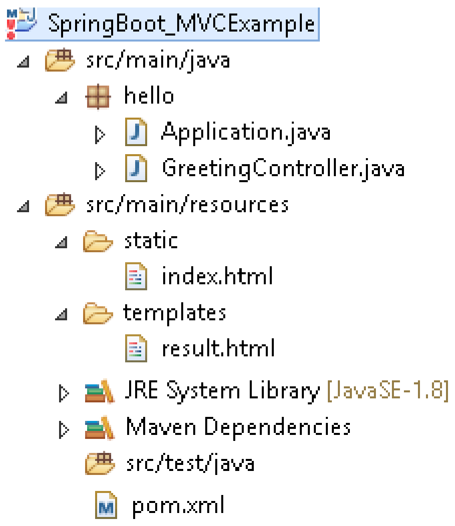
Search for dependencies: Web, Security, JPA, Actuator, Devtools.

Selected Dependencies: Web, Web Services, Jersey (JAX-RS)

Extract, import to eclipse as Existing Maven project, & **Run as→** `maven install`

If we see the Folder Structre

- **all index, welcome files must** be placed unter **→resources\static\**
- **all the result pages** must be placed under**→resources\templates\**


```
SpringBoot_MVCExample
  src/main/java
    hello
      Application.java
      GreetingController.java
  src/main/resources
    static
      index.html
    templates
      result.html
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src/test/java
  pom.xml
```

1. Choose the SpringBoot Stater Dependencies and place in pom.xml & build the project.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>SpringBoot_MVCExample</groupId>
    <artifactId>gs-serving-web-content</artifactId>
    <version>SpringBoot_MVCExample</version>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.1.RELEASE</version>
    </parent>

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

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


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

</project>
```

2. Create index.html to provide user input

```
///SpringBoot_MVCExample/src/main/resources/static/index.html
<h2>Spring Boot MVC Example</h2>
<form method="get" action="/hello">
	<h3>Enter Your name : <input type="text" name="name"></h3>
	<button type="submit">Enter</button>
</form>
```

3. Create Controller to hadle request given by user (/hello)

```
package hello;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class GreetingController {

    @RequestMapping("/hello")
    public String greeting(@RequestParam(value="name", required=false, defaultValue="World")
String name, Model model) {
        model.addAttribute("name", name);
        return "result";
    }
}
```

Here Model is a interface which conatins some usefull method to return result data to result page

4. Create result.html template to display the Results given by Controller

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<body>
    <h1><p th:text="'Hello, ' + ${name} + '!'" /></h1>
</body>
</html>
```

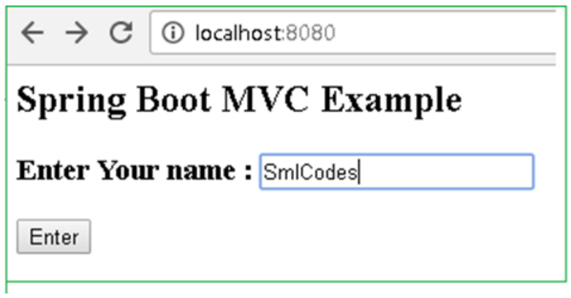
5. Create Application.java to Start & Run Spring Boot Application

```
package hello;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

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

6. Strat the Application by Run as→ `Java Application (Application.java)`

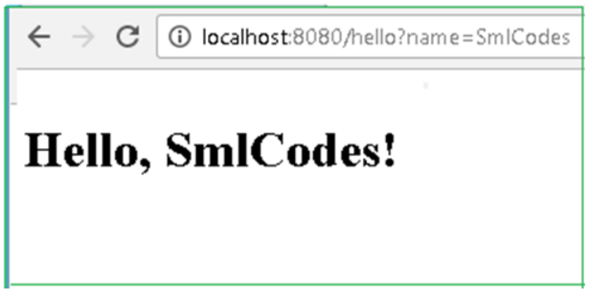
```
.   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.5.1.RELEASE)
2017-02-02TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080
```

7. Open browser, access `localhost:8080` the Output should be as below


localhost:8080

Spring Boot MVC Example

Enter Your name : SmlCodes


localhost:8080/hello?name=SmlCodes

Hello, SmlCodes!

Explanation

1. On Running Applicatio.java, Spring Boot Engine Starts and reads the all files in the projects and autowires the data and auto configures the Controller details

2. On submitting the form, Spring Boot Searchers for controller classes which are annotated with **@Controller**

3. compairs **("/hello")** path with **controller @RequestMapping("/hello"),** if matches execute the business logic method and it returns the resultpage name("**result"**)

4.SpringBoot Engine Searches the appropriate resultpage template having "**result"** as page name & displays the **result.html** page to the user

1. Model Interface

Model interface designed for adding attributes to the model. Allows for accessing the overall model as **a java.util.Map.**

Method Summary	
Model	**addAllAttributes**(Collection<?> attributeValues) Copy all attributes in the supplied Collection into this Map,
Model	**addAllAttributes**(Map<String,?> attributes) Copy all attributes in the supplied Map into this Map.
Model	**addAttribute**(Object attributeValue) Add the supplied attribute to this Map using a generated name.
Model	**addAttribute**(String attributeName, Object attributeValue) Add the supplied attribute under the supplied name.
Map<String,Object>	**asMap**() - Return the current set of model attributes as a Map.
boolean	**containsAttribute**(String attributeName) Does this model contain an attribute of the given name?
Model	**mergeAttributes(Map<String,?> attributes)**

2. Static Content

By default, Spring Boot will serve static content from a directory called **/static** (`or /public or /resources or /META-INF/resources`)

You can also customize the static resource locations using **spring.resources.static-locations** (replacing the default values with a list of directory locations).

If you do this the **default welcome page detection** will switch to your custom locations, so if there is an **index.html** in any of your locations on startup, it **will be the home page of the application.**

Spring Boot –RESTful Web Service Example

To work with webservices in SpringBoot we have to use two annotations

- **@RestController**: tells Spring Boot to consider this class as REST controller
- @**RequestMapping**: used to register paths inside it to respond to the HTTP requests.

The @RestController is a stereotype annotation. It adds @Controller and @ResponseBody annotations to the class.

@RestController = @Controller + @ResponseBody

Note - The @RestController and @RequestMapping annotations are Spring MVC annotations. They are not specific to Spring Boot.

app.controller.SpringBootRestController.java

```
package app.controller;

import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.RequestMapping;

@RestController
public class SpringBootRestController {

	@RequestMapping("/")
	public String welcome() {
		return "Spring Boot Home Page";
	}

	@RequestMapping("/hello")
	public String myData() {
		return "Smalcodes : Hello Spring Boot";
	}
}
```

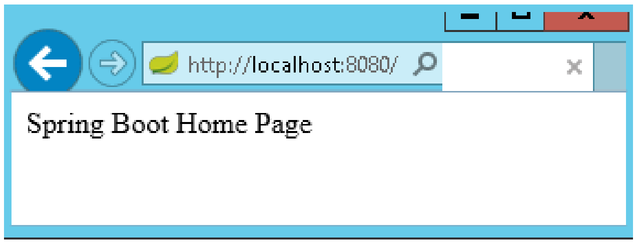
app.SpringBootApp.java

```
package app;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringBootApp {
	public static void main(String[] args) {

		SpringApplication.run(SpringBootApp.class, args);

	}
}
```

Create pom.xml same as fisrt example.

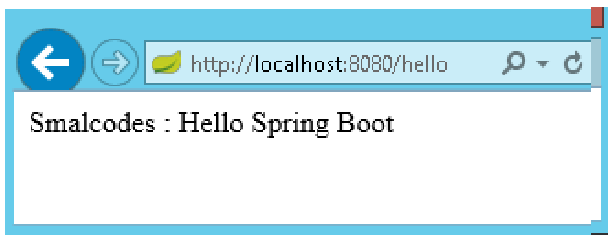
Test the Application

Right click on project > Run as >Java Application >**select SpringBootApp**



http://localhost:8080/

Spring Boot Home Page



http://localhost:8080/hello

Smalcodes : Hello Spring Boot

- In above Spring Boot main application class in ***app*** package and controller class **in *app.controller***. While starting our application, SpringBootApp class will scan all the components under that package. As we have created our controller class in **app.controller** which is inside app package, our controller was registered by spring boot.

- If you create the controller class outside of the main package, lets say com.smlcodes.*controller,* If you run the application it gives 404 error.To resolve this, we have to add ***@ComponentScan*** annotation in our Spring Boot main class, as below

```
@SpringBootApplication
@ComponentScan(basePackages="smlcodes.controller")
public class SpringBootApp {
	public static void main(String[] args) {
		SpringApplication.run(SpringBootApp.class, args);
	}
}
```

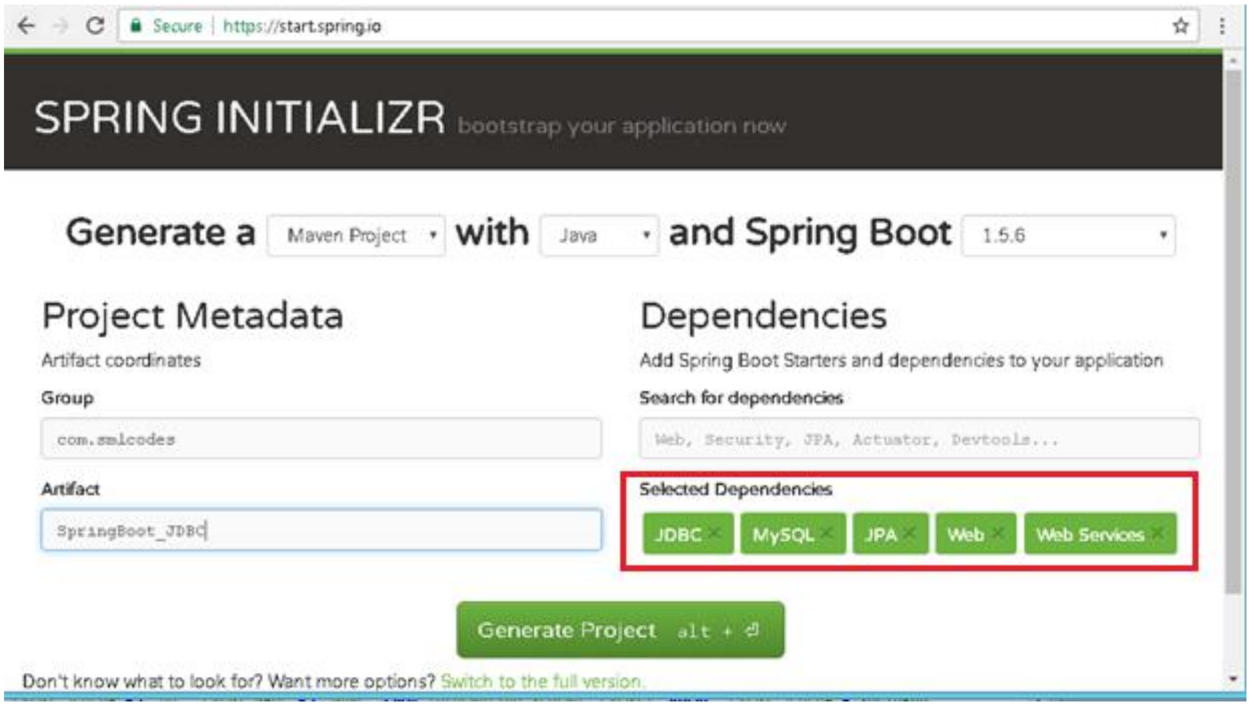
5. Spring Boot –Database

Spring Boot –JDBC Example

Spring Boot provides starter and libraries for connecting to our application with JDBC. Spring JDBC dependencies can be resolved by using either spring-boot-starter-jdbc or spring-boot-starter-data-jpa spring boot starters.

1.Create project Structure

To create project go to https://start.spring.io/ and add JDBC,MySQL,JPA dependencies to the Project.



SPRING INITIALIZR bootstrap your application now

Generate a Maven Project with Java and Spring Boot 1.5.6

Project Metadata

Artifact coordinates

Group: com.smlcodes

Artifact: SpringBoot_JDBC

Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies: Web, Security, JPA, Actuator, Devtools...

Selected Dependencies: JDBC, MySQL, JPA, Web, Web Services

Generate Project alt + ⏎

Don't know what to look for? Want more options? Switch to the full version.

Configure DataSource (application. properties)

DataSource and Connection Pool are configured in application.properties file using prefix spring.datasource. Spring boot uses javax.sql.DataSource interface to configure DataSource

```
spring.datasource.url=jdbc:mysql://localhost:3306/springdb?useSSL=false
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

Model Class(model.Student.java)

Find the MySQL table used in our example.

```
CREATE TABLE `student` (
`sno` INT(11) NOT NULL,
`name` VARCHAR(50) NULL DEFAULT NULL,
`address` VARCHAR(50) NULL DEFAULT NULL,
PRIMARY KEY (`sno`)
)
COLLATE='latin1_swedish_ci'
ENGINE=InnoDB
;
```

Create Student class with table properties

```
package app.model;

public class Student {
	private int sno;
	private String name;
	private String address;

	public Student() {
		super();
	}

	public Student(int sno, String name, String address) {
		super();
		this.sno = sno;
		this.name = name;
		this.address = address;
	}

	public int getSno() {
		return sno;
	}

	public void setSno(int sno) {
		this.sno = sno;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public String getAddress() {
		return address;
	}

	public void setAddress(String address) {
		this.address = address;
	}

}
```

DAO Class with JdbcTemplate (StudentDAO.java)

- `JdbcTemplate` is the central class to handle JDBC. It executes SQL queries and fetches their results. To use `JdbcTemplate`.
- `JdbcTemplate` dependency injection using `@Autowired` with constructor.

```
package app.dao;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;
import app.model.Student;

@Repository
public class StudentDAO {

       @Autowired
       private JdbcTemplate template;

       public List<Student> findAll() {

       List<Student> result = template.query("SELECT sno,name, address FROM Student", new
StudentRowMapper());
              return result;
       }

       public void addStudent(int sno, String name, String address) {
template.update("INSERT INTO Student(sno,name, address) VALUES (?,?,?)", sno, name, address);
       }

}
```

RowMapper Class

Spring JDBC provides `RowMapper` interface that is used to map row with a java object. We need to create our own class implementing `RowMapper` interface to map row with java object. Find the sample code to implement `RowMapper`interface.

```
package app.dao;
import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.jdbc.core.RowMapper;
import app.model.Student;

public class StudentRowMapper implements RowMapper<Student> {
       @Override
       public Student mapRow(ResultSet rs, int rowno) throws SQLException {
              // TODO Auto-generated method stub
              Student s = new Student();
              s.setSno(rs.getInt("sno"));
              s.setName(rs.getString("name"));
              s.setAddress(rs.getString("address"));
              return s;
       }
}
```

SpringBootJdbcController.java

```java
package app.controller;
import org.springframework.web.bind.annotation.RestController;
import app.dao.StudentDAO;
import app.model.Student;
import java.util.Iterator;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

@RestController
public class SpringBootJDBCController {

	@Autowired
	private StudentDAO dao;

	@RequestMapping("/jdbc")
	public String welcome() {
		return "Spring Boot Home Page";
	}

	@RequestMapping("/insert")
	public String insert(@RequestParam("sno") int sno, @RequestParam("name") String name,
			@RequestParam("address") String adr) {
		System.out.println(" ************** Inside Method ************");

		dao.addStudent(sno, name, adr);
		return "Data Inserted";
	}

	@RequestMapping("/select")
	public String select() {
		String result="";
		List<Student> list = dao.findAll();
		Iterator<Student> itr = list.iterator();
		while (itr.hasNext()) {
			Student s = (Student) itr.next();
			result = result+ s.getSno()+", ";
			result = result+ s.getName()+", ";
			result = result+ s.getAddress()+" <br>";
		}
		System.out.println("Result : "+result);
		return result;
	}
}
```

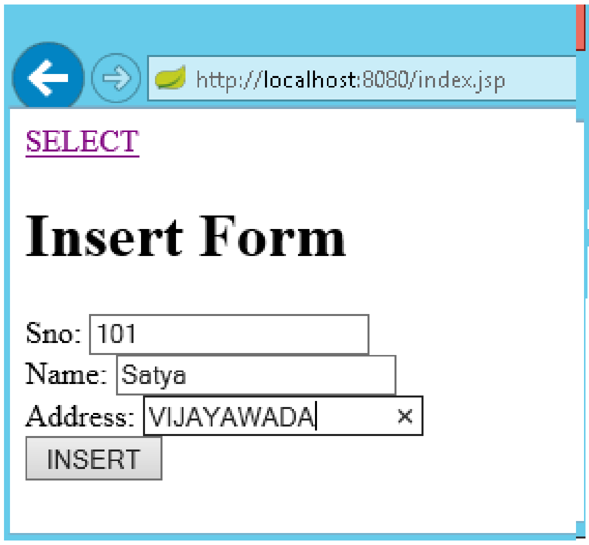
SpringBootApp.java

```java
package app;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

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

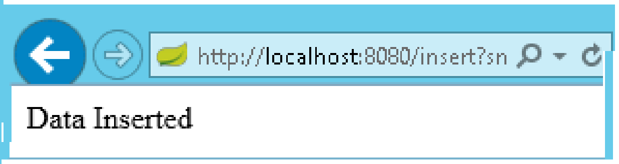
Static/index.jsp

```
<a href="/select">SELECT</a><br />

<h1>Insert Form</h1>
<form action="/insert">
        Sno: <input name="sno" type="text" /> <br>
        Name: <input name="name" type="text" /> <br>
        Address: <input name="address" type="text" /> <br>
        <input type="submit" value="INSERT" /> <br>
</form>
</body>
</html>
```

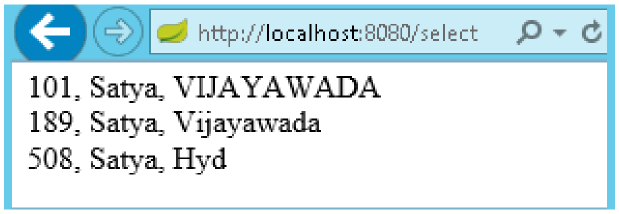
Rightclikc on Project> Runas> Java Application



http://localhost:8080/insert?sno=101 &name=Satya&address=Vijayawada



```
Data Inserted
```



```
101, Satya, VIJAYAWADA
189, Satya, Vijayawada
508, Satya, Hyd
```

We can discard RowMapper class if we write following code in StudentDAO class it self.

```
@Repository
public class StudentDAO {

        @Autowired
        private JdbcTemplate template;

        public List<Student> findAll() {

                List<Student> result = template.query("SELECT sno,name, address FROM Student",
                               (rs, rowNum) -> new Student(rs.getInt("sno"),
                                rs.getString("name"), rs.getString("address")));
                return result;

        }

        public void addStudent(int sno, String name, String address) {
                template.update("INSERT INTO Student(sno,name, address) VALUES (?,?,?)",
                 sno, name, address);
        }
}
```

Spring Boot –JPA Example

Spring Boot provides **spring-boot-starter-data-jpa** starter to connect Spring application with relational database efficiently. You can use it into project POM (Project Object Model) file.

JPA Annotations

By default, each field is mapped to a column with the name of the field. You can change the default name via **@Column (name="newColumnName").**

The following annotations can be used.

@Entity	Marks java class to a Table name
@Table(name="tabname")	Provides table name, when table name & class names are different .
@Id	**Identifies the unique ID of the database entry**
@GeneratedValue	Together with an ID this annotation defines that value is generated automatically.
@Transient	Field will not be saved in database

The central interface in Spring Data repository abstraction is **Repository** (probably not that much of a surprise). It takes the domain class to manage as well as the id type of the domain class as type arguments.

CrudRepository

The **CrudRepository** provides sophisticated CRUD functionality for the entity class that is being managed

```
public interface CrudRepository<T, ID extends Serializable> extends Repository<T, ID> {

       <S extends T> S save(S entity);
       T findOne(ID primaryKey);
       Iterable<T> findAll();
       Long count();
       void delete(T entity);
       boolean exists(ID primaryKey);
       // … more functionality omitted.
}
```

PagingAndSortingRepository

On top of the **`CrudRepository`** there is a `PagingAndSortingRepository` abstraction that adds additional methods to ease paginated access to entities:

```
public interface PagingAndSortingRepository<T, ID extends Serializable>
  extends CrudRepository<T, ID> {
       Iterable<T> findAll(Sort sort);
       Page<T> findAll(Pageable pageable);
}
```

1.Entity class : Student.java

1. create an entity class that contains the information of a single Student entry

```
package app.entity;

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

@Entity
@Table(name = "student")
public class Student {

	@Id
	@GeneratedValue(strategy = GenerationType.AUTO)
	private int sno;

	@Column(name = "name")
	private String name;

	@Column(name = "address")
	private String address;

	public Student() {
		super();
	}

	public Student(int sno, String name, String address) {
		super();
		this.sno = sno;
		this.name = name;
		this.address = address;
	}

	//Setters & getters
}
```

StudentRepository.java

We can create the repository that provides CRUD operations for ***Student*** objects by using one of the following methods:

1. Create an interface that extends the ***CrudRepository*** interface.
2. Create an interface that extends the ***Repository*** interface and add the required methods to the created interface.

```
package app.repository;
import org.springframework.data.repository.CrudRepository;
import app.entity.Student;

public interface StudentRepository extends CrudRepository<Student, String>{

}
```

StudentService.java

```
package app.service;

@Service
public class StudentService {

        @Autowired
        private StudentRepository repository;

        public List<Student> getAllStudents() {
                List<Student> studentRecords = new ArrayList<>();
                repository.findAll().forEach(studentRecords::add);
                return studentRecords;
        }

        public Student getStudent(String id) {
                return repository.findOne(id);
        }

        public void addStudent(Student studentRecord) {
                repository.save(studentRecord);
        }

        public void delete(String id) {
                repository.delete(id);
        }
}
```

StudentController.java

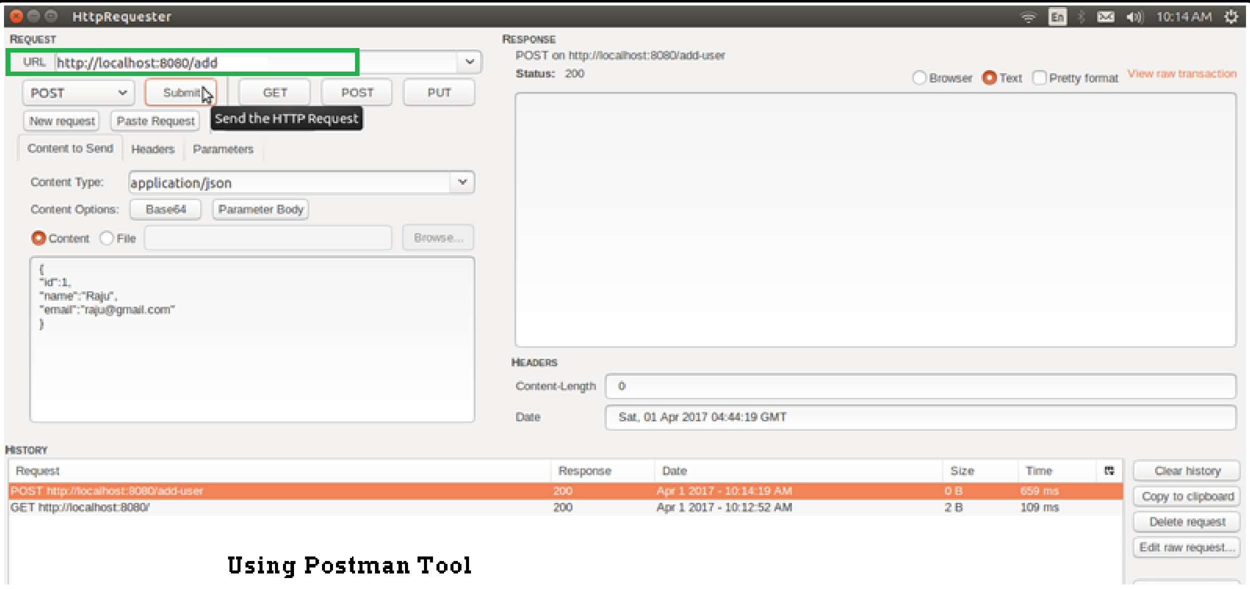
```
package app.controller;

@RestController
public class StudentController {
        @Autowired
        private StudentService studentService;

        @RequestMapping("/")
        public List<Student> getAllStudent() {
                return studentService.getAllStudents();
        }

        @RequestMapping(value = "/add", method = RequestMethod.POST)
        public void addStudent(@RequestBody Student student) {
                studentService.addStudent(student);
        }

        @RequestMapping(value = "/get/{id}", method = RequestMethod.GET)
        public Student getStudent(@PathVariable String id) {
                return studentService.getStudent(id);
        }
}
```

SpringBootApp.java

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

http://localhost:8080/ -get All Srudents

```
[
	{
		"sno": 189,
		"name": "Satya",
		"address": "Vijayawada"
	},
	{
		"sno": 508,
		"name": "Satya",
		"address": "Hyd"
	}
]
```



URL: http://localhost:8080/add

Method: POST

Content Type: application/json

[✓ Content] File

```
{
"id":1,
"name":"Raju",
"email":"raju@gmail.com"
}
```

POST on http://localhost:8080/add-user

Status: 200

Content-Length: 0

Date: Sat, 01 Apr 2017 04:44:19 GMT

Request	Response	Date	Size	Time
POST http://localhost:8080/add-user	200	Apr 1 2017 - 10:14:19 AM	0 B	659 ms
GET http://localhost:8080/	200	Apr 1 2017 - 10:12:52 AM	2 B	109 ms

Using Postman Tool

JpaRepository

JpaRepository provides some JPA related method such as flushing the persistence context and delete record in a batch. JpaRepository extends PagingAndSortingRepository which in turn extends CrudRepository.

Their main functions are:

- **CrudRepository** mainly provides CRUD functions.
- **PagingAndSortingRepository** provide methods to do pagination and sorting records.
- **JpaRepository** provides some JPA related method such as flushing the persistence context and delete record in a batch.

Because of the inheritance mentioned above, **JpaRepository** will have all the functions of **CrudRepository** and **PagingAndSortingRepository**.

Custom Queries

Spring Data JPA provides **three different approaches for creating custom queries** with query methods. Each of these approaches is described in following.

Using Method Name

- Spring Data JPA has a built in query creation mechanism which can be used for parsing queries straight from the method name of a query method.
- The method names of your repository interface are created **by combining the property names of an entity object and the supported keywords.**

```
public interface PersonRepository extends Repository<User, Long> {

        List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);

        // Enables the distinct flag for the query
List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);
List<Person> findPeopleDistinctByLastnameOrFirstname(String lastname, String firstname);

        // Enabling ignoring case for an individual property
        List<Person> findByLastnameIgnoreCase(String lastname);

        // Enabling ignoring case for all suitable properties
List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);

        // Enabling static ORDER BY for a query
        List<Person> findByLastnameOrderByFirstnameAsc(String lastname);
        List<Person> findByLastnameOrderByFirstnameDesc(String lastname);
}
```

JPA Named Queries

Spring Data JPA provides also support for the JPA Named Queries. You have got following alternatives for declaring the named queries:

- You can use either ***named-query*** **XML** element or ***@NamedQuery*** annotation to create named queries with the JPA query language.

- You can use either ***named-native-query*** XML element **or *@NamedNative*** query annotation to create queries with SQL if you are ready to tie your application with a specific database platform.

The only thing you have to do to use the created named queries is to name the query method of your repository interface to match with the name of your named query. See below Example code

```
@Entity
@NamedQuery(name = "Person.findByName", query = "SELECT p FROM Person p WHERE
LOWER(p.lastName) = LOWER(?1)")
@Table(name = "persons")
public class Person {

        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
        private Long id;
        @Column(name = "creation_time", nullable = false)
        private Date creationTime;

        @Column(name = "first_name", nullable = false)
        private String firstName;
}
```

The relevant part of my ***PersonRepository*** interface looks following

```
public interface PersonRepository extends JpaRepository<Person, Long> {
    //A list of persons whose last name is an exact match with the given last name.
        public List<Person> findByName(String lastName);
}
```

@Query Annotation

- The ***@Query*** annotation can be used to create queries by using the JPA query language and to **bind these queries directly to the methods of your repository interface**.
- When the query method is called, Spring Data **JPA will execute the query specified by the *@Query* annotation**
- If there is a collision between the *@Query* annotation and the named queries, the query specified by using *@Query* annotation will be executed

```
public interface ProductRepository
   extends CrudRepository<Product, Long> {
    @Query("FROM Product")
    List<Product> findAllProducts();
}
```

You may use positional parameters instead of named parameters in queries. Positional parameters are prefixed with a question mark **(?)** followed the numeric position of the parameter in the query. The `Query.setParameter(integer position, Object value)` method is used to set the parameter values.

```
public List findWithName(String name) {
    return em.createQuery(
        "SELECT c FROM Customer c WHERE c.name LIKE ?1")
        .setParameter(1, name)
        .getResultList();
}
```

Automatic Query Generation

The **<jpa:repositories/>** has an option query-lookup-strategy which defaults to "**create-if-not-found" which will generate queries for us.**The default is "create-if-not-found". Other options are "create" or "use-declared-query".

```
<jpa:repositories base-package="com.gordondickens.myapp.repository"
     query-lookup-strategy="create-if-not-found"/>
```

To create a find method that effectively does `@Query("FROM Product p where p.productId =:productId")`

```
public interface ProductRepository extends CrudRepository<Product, Long> {
   ...

   @Query
   Product findByProductId(String productId);

   ...
```

Example

Student.java

```
package app.entity;

@Entity
@Table(name = "student")
public class Student {

        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
        private int sno;

        @Column(name = "name")
        private String name;

        @Column(name = "address")
        private String address;

        @Override
        public String toString() {
        String str = "Student[" + "Sno: " + getSno() + ", Name:" + getName() + ", " +
"Address : " + getAddress() + "]";
                return str;
        }
//Setters & getters
}
```

StudentRepository.java

```
public interface StudentRepository extends CrudRepository<Student, Long> {

        List<Student> findBySno(int sno);

        List<Student> findByName(String name);

        // custom query example and return a stream
        @Query("select c from Student c where c.address = :address")
        Stream<Student> findByAddress(@Param("address") String address);
}
```

Application.java

```java
package app;

@SpringBootApplication
public class Application implements CommandLineRunner {

	@Autowired
	DataSource dataSource;

	@Autowired
	StudentRepository repository;

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

	@Transactional(readOnly = true)
	@Override
	public void run(String... args) throws Exception {

		System.out.println("DATASOURCE = " + dataSource);

		System.out.println("\n1.findAll()...");
		for (Student student : repository.findAll()) {
			System.out.println(student);
		}

		System.out.println("\n2.findByName(String name)...");
		for (Student student : repository.findByName("Satya")) {
			System.out.println(student);
		}

		System.out.println("\n3.findByAddress(@Param(\"name\"))...");
		try (Stream<Student> s =repository.findByAddress("Vijayawada")) {
			s.forEach(x -> System.out.println(x));
			System.out.println("Done!");
			exit(0);
		}
	}
}
```

```
1.findAll()...
Student[Sno: 0, Name:null, Address : null]
Student[Sno: 101, Name:Satya, Address : VIJAYAWADA]
Student[Sno: 102, Name:Satya, Address : Vijayawada]
Student[Sno: 147, Name:kumar, Address : Hyderabad]
Student[Sno: 189, Name:Satya, Address : Vijayawada]
Student[Sno: 508, Name:Satya, Address : Hyd]

2.findByName(String name)...
Student[Sno: 101, Name:Satya, Address : VIJAYAWADA]
Student[Sno: 102, Name:Satya, Address : Vijayawada]
Student[Sno: 189, Name:Satya, Address : Vijayawada]
Student[Sno: 508, Name:Satya, Address : Hyd]

4.findByAddress(@Param("name") String name)...
Student[Sno: 101, Name:Satya, Address : VIJAYAWADA]
Student[Sno: 102, Name:Satya, Address : Vijayawada]
Student[Sno: 189, Name:Satya, Address : Vijayawada]
```

Spring Boot –MongoDB REST Example

Configuration file application.properties

```
# Create new database : ‘smlcodes’
spring.data.mongodb.database=smlcodes
spring.data.mongodb.host=localhost
spring.data.mongodb.port=27017
```

We need to model our documents. Let’s call ours ‘**Booking**’ and give it a make, model, and description. Here is our Java class to accomplish this

```
package smlcodes.repository;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

@Document
public class Student{

	@Id
	String sno;
	String name;
	String address;
	public String getSno() {
		return sno;
	}
	public void setSno(String sno) {
		this.sno = sno;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public String getAddress() {
		return address;
	}
	public void setAddress(String address) {
		this.address = address;
	}
}
```

- **@Id-** id provided by Mongo for a document.
- **@Document-** provides a collection name.

BookingRepository.java

The **MongoRepository** provides basic CRUD operation methods and also an API to find all documents in the collection.

```
@Transactional
public interface StudentRepository extends MongoRepository<Student, String> {
	public Student findBySno(int sno);
}
```

BookingController.java

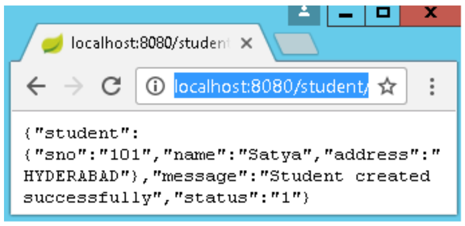
```java
package smlcodes.controller;

@RestController
@RequestMapping("/student")
public class StudentController {

        @Autowired
        StudentRepository studentRepository;

        @RequestMapping("/create")
        public Map<String, Object> create(Student student) {
                student = studentRepository.save(student);
                Map<String, Object> dataMap = new HashMap<String, Object>();
                dataMap.put("message", "Student created successfully");
                dataMap.put("status", "1");
                dataMap.put("student", student);
            return dataMap;
        }

        @RequestMapping("/read")
        public Map<String, Object> read(@RequestParam int sno) {
                Student student = studentRepository.findBySno(sno);
                Map<String, Object> dataMap = new HashMap<String, Object>();
                dataMap.put("message", "Student found successfully");
                dataMap.put("status", "1");
                dataMap.put("student", student);
            return dataMap;
        }

        @RequestMapping("/readall")
        public Map<String, Object> readAll() {
                List<Student> students = studentRepository.findAll();
                Map<String, Object> dataMap = new HashMap<String, Object>();
                dataMap.put("message", "Student found successfully");
                dataMap.put("totalStudent", students.size());
                dataMap.put("status", "1");
                dataMap.put("students", students);
            return dataMap;
        }
}
```

SpringBootMongoDbApplication.java

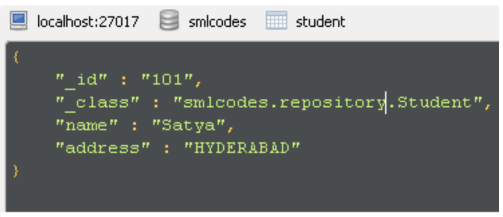
```java
@SpringBootApplication
public class SpringBootMongoDbApplication {
        public static void main(String[] args) {
                SpringApplication.run(SpringBootMongoDbApplication.class, args);
        }
}
```

Test

http://localhost:8080/student/create?sno=101&name=Satya&address=HYDERABAD



```
{"student":
{"sno":"101","name":"Satya","address":"
HYDERABAD"},"message":"Student created
successfully","status":"1"}
```



```
{
    "_id" : "101",
    "_class" : "smlcodes.repository.Student",
    "name" : "Satya",
    "address" : "HYDERABAD"
}
```

References

- http://sivalabs.in/2016/03/springboot-working-with-jdbctemplate/
- https://www.javatpoint.com/spring-boot-jpa
- https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-part-two-crud/
- https://www.dineshonjava.com/spring-boot-and-mongodb-in-rest-application/
- https://spring.io/guides/gs/spring-boot/#scratch
- http://docs.spring.io/autorepo/docs/spring-boot/current/reference/html/(best)
- http://www.dineshonjava.com/2016/06/introduction-to-spring-boot-a-spring-boot-complete-guide.html#.WI7wB1N965t
- http://websystique.com/spring-boot-tutorial/
- https://www.mkyong.com/tag/spring-boot/
- (Best)Helloworld : http://www.shristitechlabs.com/introduction-to-spring-boot/
- https://www.mkyong.com/spring-boot/spring-boot-spring-data-mongodb-example/
- https://tests4geeks.com/spring-data-boot-mongodb-example/
- https://avaldes.com/building-a-realtime-angularjs-dashboard-using-spring-rest-and-mongodb-part-1/
- Final:https://www.callicoder.com/spring-boot-mongodb-angular-js-rest-api-tutorial/