



# Spring Boot



**tutorialspoint**

SIMPLY EASY LEARNING

[www.tutorialspoint.com](http://www.tutorialspoint.com)



<https://www.facebook.com/tutorialspointindia>



<https://twitter.com/tutorialspoint>

## About the Tutorial

---

Spring Boot is an open source Java-based framework used to create a Micro Service. It is developed by Pivotal Team. It is easy to create a stand-alone and production ready spring applications using Spring Boot. Spring Boot contains a comprehensive infrastructure support for developing a microservice and enables you to develop enterprise-ready applications that you can “**just run**”.

## Audience

---

This tutorial is designed for Java developers to understand and develop production-ready spring applications with minimum configurations. It explores major features of Spring Boot such as Starters, Auto-configuration, Beans, Actuator and more.

By the end of this tutorial, you will gain an intermediate level of expertise in Spring Boot.

## Prerequisites

---

This tutorial is written for readers who have a prior experience of Java, Spring, Maven, and Gradle. You can easily understand the concepts of Spring Boot if you have knowledge on these concepts. It would be an additional advantage if you have an idea about writing a RESTful Web Service. If you are a beginner, we suggest you to go through tutorials related to these concepts before you start with Spring Boot.

## Copyright and Disclaimer

---

© Copyright 2018 by Tutorials Point (I) Pvt. Ltd.

All the content and graphics published in this e-book are the property of Tutorials Point (I) Pvt. Ltd. The user of this e-book is prohibited to reuse, retain, copy, distribute or republish any contents or a part of contents of this e-book in any manner without written consent of the publisher.

We strive to update the contents of our website and tutorials as timely and as precisely as possible, however, the contents may contain inaccuracies or errors. Tutorials Point (I) Pvt. Ltd. provides no guarantee regarding the accuracy, timeliness or completeness of our website or its contents including this tutorial. If you discover any errors on our website or in this tutorial, please notify us at [contact@tutorialspoint.com](mailto:contact@tutorialspoint.com)

## Table of Contents

---

About the Tutorial .....	i
Audience.....	i
Prerequisites.....	i
Copyright and Disclaimer .....	i
Table of Contents.....	ii
 1. SPRING BOOT – INTRODUCTION .....	 1
What is Micro Service?.....	1
What is Spring Boot?.....	1
Why Spring Boot? .....	2
How does it work? .....	2
Spring Boot Starters .....	2
Auto Configuration .....	3
Spring Boot Application .....	4
Component Scan.....	4
 2. SPRING BOOT – QUICK START .....	 6
Prerequisites.....	6
Spring Boot CLI.....	6
 3. SPRING BOOT – BOOTSTRAPPING .....	 8
Spring Initializer .....	8
Maven.....	9
Gradle.....	10
Class Path Dependencies .....	11
Main Method.....	12
Write a Rest Endpoint.....	12
Create an Executable JAR.....	13

Run Hello World with Java .....	14
4. SPRING BOOT – TOMCAT DEPLOYMENT .....	16
Spring Boot Servlet Initializer .....	16
Setting Main Class .....	17
Update packaging JAR into WAR .....	17
Packaging your Application .....	19
Deploy into Tomcat .....	20
5. SPRING BOOT – BUILD SYSTEMS .....	25
Dependency Management .....	25
Maven Dependency .....	25
Gradle Dependency .....	26
6. SPRING BOOT – CODE STRUCTURE .....	27
Default package .....	27
Typical Layout .....	27
6. SPRING BOOT – CODE STRUCTURE .....	27
7. SPRING BOOT – SPRING BEANS AND DEPENDENCY INJECTION .....	28
8. SPRING BOOT – RUNNERS .....	29
Application Runner .....	29
Command Line Runner .....	30
9. SPRING BOOT – APPLICATION PROPERTIES .....	31
Command Line Properties .....	31
Properties File .....	31
YAML File .....	31
Externalized Properties .....	32
Use of @Value Annotation .....	32

Spring Boot Active Profile .....	33
10. SPRING BOOT – LOGGING .....	37
Log Format.....	37
Console Log Output.....	37
File Log Output .....	37
Log Levels .....	38
Configure Logback.....	38
11. SPRING BOOT – BUILDING RESTFUL WEB SERVICES .....	41
Rest Controller.....	43
Request Mapping.....	43
Request Body.....	44
Path Variable .....	44
Request Parameter .....	44
GET API .....	44
POST API .....	46
PUT API.....	47
DELETE API.....	48
12. SPRING BOOT – EXCEPTION HANDLING .....	54
Controller Advice .....	54
ExceptionHandler.....	54
13. SPRING BOOT – INTERCEPTOR .....	62
14. SPRING BOOT – SERVLET FILTER.....	71
15. SPRING BOOT – TOMCAT PORT NUMBER .....	77
Custom Port.....	77
Random Port.....	77

16. SPRING BOOT – REST TEMPLATE.....	78
GET .....	79
POST .....	80
PUT .....	81
DELETE .....	82
17. SPRING BOOT – FILE HANDLING.....	89
File Upload.....	89
File Download .....	90
18. SPRING BOOT – SERVICE COMPONENTS .....	96
19. SPRING BOOT – THYMELEAF .....	106
Thymeleaf Templates.....	106
Web Application .....	106
20. SPRING BOOT – CONSUMING RESTFUL WEB SERVICES.....	113
Angular JS .....	122
21. SPRING BOOT – CORS SUPPORT .....	124
Enable CORS in Controller Method .....	124
Global CORS Configuration .....	124
22. SPRING BOOT – INTERNATIONALIZATION .....	126
Dependencies .....	126
LocaleResolver .....	126
LocaleChangeInterceptor .....	127
Messages Sources .....	127
HTML file .....	128
23. SPRING BOOT – SCHEDULING .....	134
Java Cron Expression.....	134

Fixed Rate .....	135
Fixed Delay .....	136
24. SPRING BOOT – ENABLING HTTPS .....	138
Self-Signed Certificate .....	138
Configure HTTPS .....	139
25. SPRING BOOT – EUREKA SERVER .....	140
Building a Eureka Server .....	140
26. SPRING BOOT – SERVICE REGISTRATION WITH EUREKA .....	146
27. SPRING BOOT – ZUUL PROXY SERVER AND ROUTING .....	153
Creating Zuul Server Application .....	153
28. SPRING BOOT – SPRING CLOUD CONFIGURATION SERVER .....	160
Creating Spring Cloud Configuration Server .....	160
29. SPRING BOOT – SPRING CLOUD CONFIGURATION CLIENT .....	166
Working with Spring Cloud Configuration Server .....	166
30. SPRING BOOT – ACTUATOR .....	169
Enabling Spring Boot Actuator .....	169
31. SPRING BOOT – ADMIN SERVER .....	171
32. SPRING BOOT – ADMIN CLIENT .....	176
33. SPRING BOOT – ENABLING SWAGGER2 .....	179
34. SPRING BOOT – CREATING DOCKER IMAGE .....	186
Create Dockerfile .....	186
Maven .....	186
Gradle .....	190
35. SPRING BOOT – TRACING MICRO SERVICE LOGS .....	194

Spring Cloud Sleuth .....	194
Zipkin Server .....	199
36. SPRING BOOT – FLYWAY DATABASE.....	206
Configuring Flyway Database .....	206
37. SPRING BOOT – SENDING EMAIL.....	213
38. SPRING BOOT – HYSTRIX .....	219
39. SPRING BOOT – WEB SOCKET.....	226
40. SPRING BOOT – BATCH SERVICE.....	235
41. SPRING BOOT – SPRING FOR APACHE KAFKA .....	244
Producing Messages .....	244
Consuming a Message.....	245
42. SPRING BOOT – TWILIO.....	252
Sending SMS .....	252
Voice Calls.....	257
43. SPRING BOOT – UNIT TEST CASES .....	262
Mockito .....	262
44. SPRING BOOT – REST CONTROLLER UNIT TEST .....	269
Writing a Unit Test for REST Controller .....	269
45. SPRING BOOT – DATABASE HANDLING .....	276
Connect to H2 database .....	276
Connect MySQL.....	277
Connect Redis .....	279
JdbcTemplate.....	280
Multiple DataSource .....	280



46. SPRING BOOT – SECURING WEB APPLICATIONS.....	284
Securing a Web application.....	284
47. SPRING BOOT SECURITY – OAUTH2 WITH JWT .....	294
Authorization Server .....	294
Resource Server .....	294
OAuth2 .....	294
JWT Token .....	294
48. SPRING BOOT – GOOGLE CLOUD PLATFORM .....	311
Google Cloud SQL.....	314
49. SPRING BOOT – GOOGLE OAUTH2 SIGN-IN.....	316

# 1. Spring Boot – Introduction

Spring Boot is an open source Java-based framework used to create a micro Service. It is developed by Pivotal Team and is used to build stand-alone and production ready spring applications. This chapter will give you an introduction to Spring Boot and familiarizes you with its basic concepts.

## What is Micro Service?

---

Micro Service is an architecture that allows the developers to develop and deploy services independently. Each service running has its own process and this achieves the lightweight model to support business applications.

### Advantages

Micro services offers the following advantages to its developers:

- Easy deployment
- Simple scalability
- Compatible with Containers
- Minimum configuration
- Lesser production time

## What is Spring Boot?

---

Spring Boot provides a good platform for Java developers to develop a stand-alone and production-grade spring application that you can **just run**. You can get started with minimum configurations without the need for an entire Spring configuration setup.

### Advantages

Spring Boot offers the following advantages to its developers:

- Easy to understand and develop spring applications
- Increases productivity
- Reduces the development time

### Goals

Spring Boot is designed with the following goals:

- To avoid complex XML configuration in Spring
- To develop a production ready Spring applications in an easier way
- To reduce the development time and run the application independently
- Offer an easier way of getting started with the application

## Why Spring Boot?

---

You can choose Spring Boot because of the features and benefits it offers as given here:

- It provides a flexible way to configure Java Beans, XML configurations, and Database Transactions.
- It provides a powerful batch processing and manages REST endpoints.
- In Spring Boot, everything is auto configured; no manual configurations are needed.
- It offers annotation-based spring application
- Eases dependency management
- It includes Embedded Servlet Container

## How does it work?

---

Spring Boot automatically configures your application based on the dependencies you have added to the project by using **@EnableAutoConfiguration** annotation. For example, if MySQL database is on your classpath, but you have not configured any database connection, then Spring Boot auto-configures an in-memory database.

The entry point of the spring boot application is the class contains **@SpringBootApplication** annotation and the main method.

Spring Boot automatically scans all the components included in the project by using **@ComponentScan** annotation.

## Spring Boot Starters

---

Handling dependency management is a difficult task for big projects. Spring Boot resolves this problem by providing a set of dependencies for developers convenience.

For example, if you want to use Spring and JPA for database access, it is sufficient if you include **spring-boot-starter-data-jpa** dependency in your project.

Note that all Spring Boot starters follow the same naming pattern **spring-boot-starter-\***, where \* indicates that it is a type of the application.

## Examples

Look at the following Spring Boot starters explained below for a better understanding:

**Spring Boot Starter Actuator dependency** is used to monitor and manage your application. Its code is shown below:

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

**Spring Boot Starter Security dependency** is used for Spring Security. Its code is shown below:

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

**Spring Boot Starter web dependency** is used to write a Rest Endpoints. Its code is shown below:

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

**Spring Boot Starter Thyme Leaf dependency** is used to create a web application. Its code is shown below:

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

**Spring Boot Starter Test dependency** is used for writing Test cases. Its code is shown below:

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

## Auto Configuration

Spring Boot Auto Configuration automatically configures your Spring application based on the JAR dependencies you added in the project. For example, if MySQL database is on your class path, but you have not configured any database connection, then Spring Boot auto-configures an in-memory database.

For this purpose, you need to add **@EnableAutoConfiguration** annotation or **@SpringBootApplication** annotation to your main class file. Then, your Spring Boot application will be automatically configured.

Observe the following code for a better understanding:

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;

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

## Spring Boot Application

The entry point of the Spring Boot Application is the class contains **@SpringBootApplication** annotation. This class should have the main method to run the Spring Boot application. **@SpringBootApplication** annotation includes Auto-Configuration, Component Scan, and Spring Boot Configuration.

If you added **@SpringBootApplication** annotation to the class, you do not need to add the **@EnableAutoConfiguration**, **@ComponentScan** and **@SpringBootConfiguration** annotation. The **@SpringBootApplication** annotation includes all other annotations.

Observe the following code for a better understanding:

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

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

## Component Scan

Spring Boot application scans all the beans and package declarations when the application initializes. You need to add the **@ComponentScan** annotation for your class file to scan your components added in your project.

Observe the following code for a better understanding:

```
import org.springframework.boot.SpringApplication;
```

```
import org.springframework.context.annotation.ComponentScan;

@ComponentScan
public class DemoApplication {

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

## 2. Spring Boot – Quick Start

This chapter will teach you how to create a Spring Boot application using Maven and Gradle.

### Prerequisites

---

Your system need to have the following minimum requirements to create a Spring Boot application:

- Java 7
- Maven 3.2
- Gradle 2.5

### Spring Boot CLI

---

The Spring Boot CLI is a command line tool and it allows us to run the Groovy scripts. This is the easiest way to create a Spring Boot application by using the Spring Boot Command Line Interface. You can create, run and test the application in command prompt itself.

This section explains you the steps involved in manual installation of Spring Boot CLI . For further help, you can use the following link: <https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/#getting-started-installing-spring-boot>

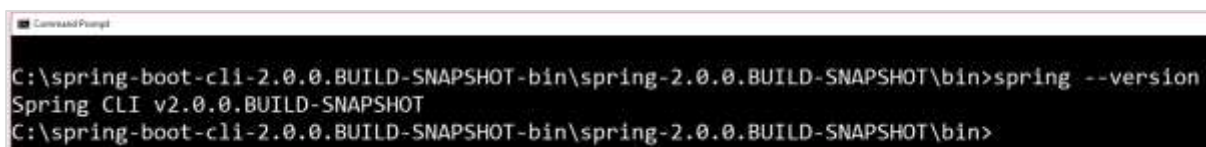
You can also download the Spring CLI distribution from the Spring Software repository at: <https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/#getting-started-manual-cli-installation>

For manual installation, you need to use the following two folders:

- **spring-boot-cli-2.0.0.BUILD-SNAPSHOT-bin.zip**
- **spring-boot-cli-2.0.0.BUILD-SNAPSHOT-bin.tar.gz**

After the download, unpack the archive file and follow the steps given in the install.txt file. Not that it does not require any environment setup.

In Windows, go to the Spring Boot CLI **bin** directory in the command prompt and run the command **spring --version** to make sure spring CLI is installed correctly. After executing the command, you can see the spring CLI version as shown below:



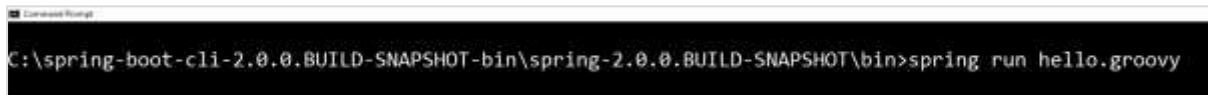
```
Command Prompt
C:\spring-boot-cli-2.0.0.BUILD-SNAPSHOT-bin\spring-2.0.0.BUILD-SNAPSHOT\bin>spring --version
Spring CLI v2.0.0.BUILD-SNAPSHOT
C:\spring-boot-cli-2.0.0.BUILD-SNAPSHOT-bin\spring-2.0.0.BUILD-SNAPSHOT\bin>
```

## Run Hello World with Groovy

Create a simple groovy file which contains the Rest Endpoint script and run the groovy file with spring boot CLI. Observe the code shown here for this purpose:

```
@Controller
class Example {
    @RequestMapping("/")
    @ResponseBody
    public String hello() {
        "Hello Spring Boot"
    }
}
```

Now, save the groovy file with the name **hello.groovy**. Note that in this example, we saved the groovy file inside the Spring Boot CLI **bin** directory. Now run the application by using the command **spring run hello.groovy** as shown in the screenshot given below:



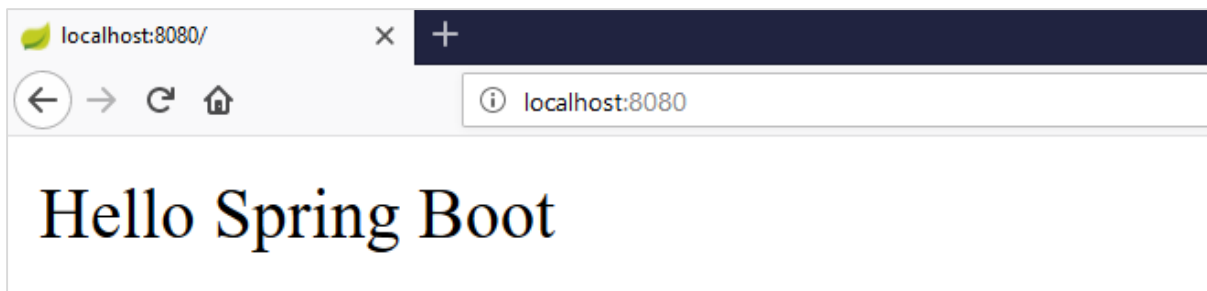
```
C:\spring-boot-cli-2.0.0.BUILD-SNAPSHOT-bin\spring-2.0.0.BUILD-SNAPSHOT\bin>spring run hello.groovy
```

Once you run the groovy file, required dependencies will download automatically and it will start the application in Tomcat 8080 port as shown in the screenshot given below:



```
2017-12-02 12:18:15.822 INFO 12788 --- [runner-0] o.s.j.e.a.AnnotationBeanExporter : Registering beans for JMX exposure on startup
2017-12-02 12:18:16.006 INFO 12788 --- [runner-0] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2017-12-02 12:18:16.015 INFO 12788 --- [runner-0] o.s.boot.SpringApplication : Started application in 12.966 seconds (JVM running for 85.283)
```

Once Tomcat starts, go to the web browser and hit the URL <http://localhost:8080/>, and you can see the output as shown.



localhost:8080/

Hello Spring Boot



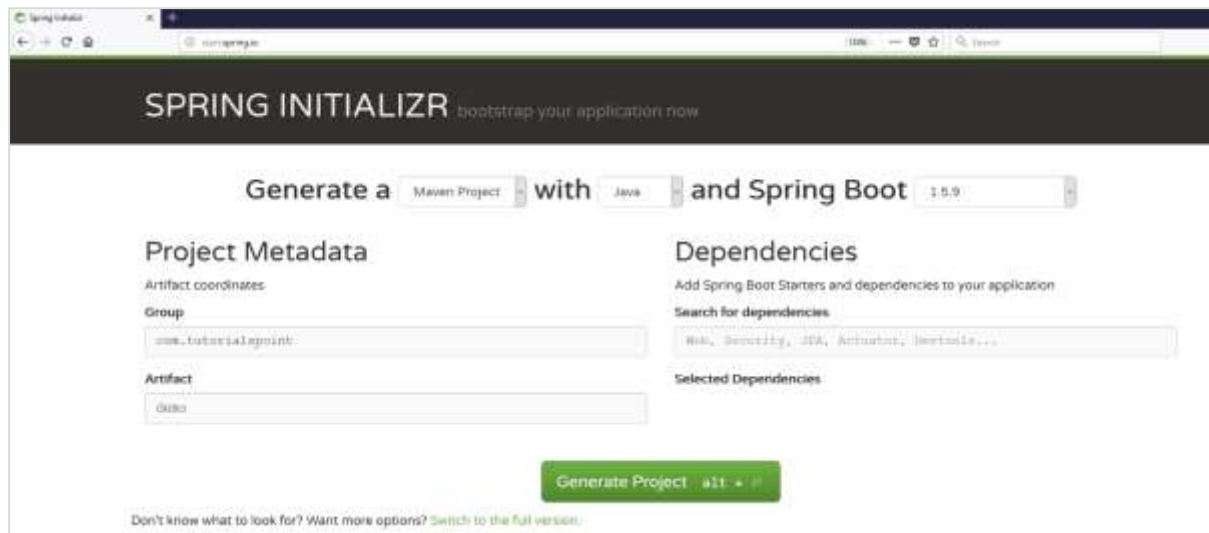
# 3. Spring Boot – Bootstrapping

This chapter will explain you how to perform bootstrapping on a Spring Boot application.

## Spring Initializer

One of the ways to Bootstrapping a Spring Boot application is by using Spring Initializer. To do this, you will have to visit the Spring Initializer web page <http://start.spring.io/> and choose your Build, Spring Boot Version and platform. Also, you need to provide a Group, Artifact and required dependencies to run the application.

Observe the following screenshot that shows an example where we added the **spring-boot-starter-web** dependency to write REST Endpoints.

The screenshot shows the Spring Initializer web application interface. At the top, it says "SPRING INITIALIZR bootstrap your application now". Below this, there's a form to generate a project. The form has three main sections: "Project Metadata", "Dependencies", and a "Generate Project" button. In the "Project Metadata" section, the "Group" field is filled with "com.tutorialspoint" and the "Artifact" field is filled with "Demo". In the "Dependencies" section, the "Search for dependencies" field is filled with "Web, Security, JPA, Actuator, Hystrix...". The "Selected Dependencies" section is empty. The "Generate Project" button is green and has the text "Generate Project" and "alt + G". At the bottom, there's a link that says "Don't know what to look for? Want more options? Switch to the full version."

Once you provided the Group, Artifact, Dependencies, Build Project, Platform and Version, click **Generate Project** button. The zip file will download and the files will be extracted.

This section explains you the examples by using both Maven and Gradle.

## Maven

After you download the project, unzip the file. Now, your **pom.xml** file looks as shown below:

```
<?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>com.tutorialspoint</groupId>
    <artifactId>demo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>demo</name>
    <description>Demo project for Spring Boot</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.8.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
    </properties>

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

```

## Gradle

Once you download the project, unzip the file. Now your **build.gradle** file looks as shown below:

```

buildscript {
    ext {
        springBootVersion = '1.5.8.RELEASE'
    }
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-
plugin:${springBootVersion}")
    }
}
apply plugin: 'java'

apply plugin: 'eclipse'

```

```

apply plugin: 'org.springframework.boot'

group = 'com.tutorialspoint'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = 1.8

repositories {
    mavenCentral()
}

dependencies {
    compile('org.springframework.boot:spring-boot-starter-web')
    testCompile('org.springframework.boot:spring-boot-starter-test')
}

```

## Class Path Dependencies

Spring Boot provides a number of **Starters** to add the jars in our class path. For example, for writing a Rest Endpoint, we need to add the **spring-boot-starter-web** dependency in our class path. Observe the codes shown below for a better understanding:

### Maven dependency

```

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

```

### Gradle dependency

```

dependencies {
    compile('org.springframework.boot:spring-boot-starter-web')
}

```

## Main Method

---

The main method should be writing the Spring Boot Application class. This class should be annotated with **@SpringBootApplication**. This is the entry point of the spring boot application to start. You can find the main class file under **src/java/main** directories with the default package.

In this example, the main class file is located at the **src/java/main** directories with the default package **com.tutorialspoint.demo**. Observe the code shown here for a better understanding:

```
package com.tutorialspoint.demo;

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

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

## Write a Rest Endpoint

---

To write a simple Hello World Rest Endpoint in the Spring Boot Application main class file itself, follow the steps shown below:

- Firstly, add the **@RestController** annotation at the top of the class.
- Now, write a Request URI method with **@RequestMapping** annotation.
- Then, the Request URI method should return the **Hello World** string.

Now, your main Spring Boot Application class file will look like as shown in the code given below:

```
package com.tutorialspoint.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

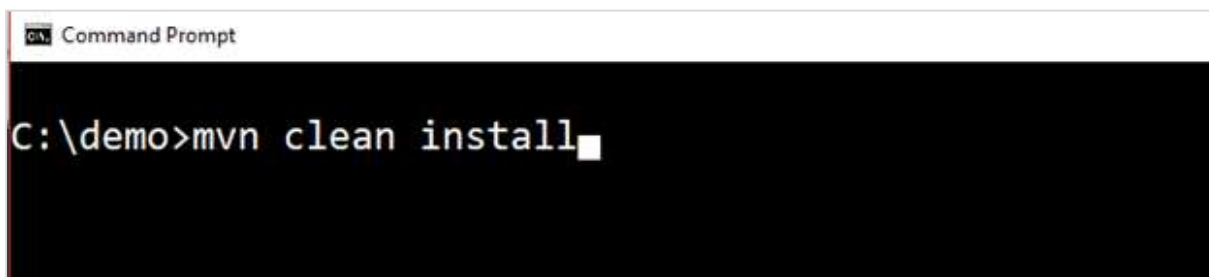
@SpringBootApplication
@RestController
public class DemoApplication {
```

```
public static void main(String[] args) {  
    SpringApplication.run(DemoApplication.class, args);  
}  
  
@RequestMapping(value="/")  
public String hello() {  
    return "Hello World";  
}  
}
```

## Create an Executable JAR

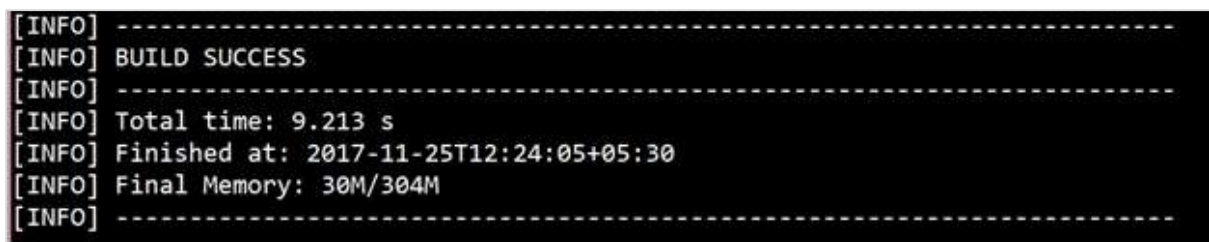
Let us create an executable JAR file to run the Spring Boot application by using Maven and Gradle commands in the command prompt as shown below:

Use the Maven command **mvn clean install** as shown below:




```
C:\demo>mvn clean install
```

After executing the command, you can see the **BUILD SUCCESS** message at the command prompt as shown below:



```
[INFO] -----  
[INFO] BUILD SUCCESS  
[INFO] -----  
[INFO] Total time: 9.213 s  
[INFO] Finished at: 2017-11-25T12:24:05+05:30  
[INFO] Final Memory: 30M/304M  
[INFO] -----
```

Use the Gradle command **gradle clean build** as shown below:



```
C:\demo>gradle clean build
```

After executing the command, you can see the **BUILD SUCCESSFUL** message in the command prompt as shown below:

```

C:\demo>gradle clean build
Starting a Gradle Daemon, 3 busy and 1 incompatible and 1 stopped Daemons could not be reused, use --status for details
:clean
:compileJava
:processResources
:classes
:findMainClass
:jar
:bootRepackage
:assemble
:compileTestJava
:processTestResources NO-SOURCE
:testClasses
:test
2017-12-02 12:24:09.098 INFO 5944 --- [ Thread-5] o.s.w.c.s.GenericWebApplicationContext : Closing org.springframework.web.context.support.GenericWebApplicationContext
text82c1411b7: startup date [Sat Dec 02 12:24:00 IST 2017]; root of context hierarchy
:check
:build

BUILD SUCCESSFUL
Total time: 31.036 secs

```

## Run Hello World with Java

Once you have created an executable JAR file, you can find it under the following directories.

For Maven, you can find the JAR file under the target directory as shown below:

```

C:\demo\target>dir
Volume in drive C has no label.
Volume Serial Number is 8CDD-0B1B

Directory of C:\demo\target

11/29/2017  10:55 AM    <DIR>          .
11/29/2017  10:55 AM    <DIR>          ..
12/02/2017  11:34 AM    <DIR>          classes
11/29/2017  10:55 AM             21,316,462 demo-0.0.1-SNAPSHOT.jar
11/29/2017  10:55 AM             16,412 demo-0.0.1-SNAPSHOT.jar.original
11/29/2017  10:55 AM    <DIR>          generated-sources
11/29/2017  10:55 AM    <DIR>          generated-test-sources
11/29/2017  10:55 AM    <DIR>          maven-archiver
11/29/2017  10:55 AM    <DIR>          maven-status
11/29/2017  10:55 AM    <DIR>          surefire-reports
12/02/2017  11:34 AM    <DIR>          test-classes
                2 File(s)      21,332,874 bytes
                9 Dir(s)   302,996,082,688 bytes free

```

For Gradle, you can find the JAR file under the **build/libs** directory as shown below:

```

Command Prompt

C:\demo\build\libs>dir
Volume in drive C has no label.
Volume Serial Number is 8CDD-0B1B

Directory of C:\demo\build\libs

11/23/2017  07:38 PM    <DIR>          .
11/23/2017  07:38 PM    <DIR>          ..
11/23/2017  07:38 PM             14,491,694 demo-0.0.1-SNAPSHOT.jar
11/23/2017  07:38 PM             1,577 demo-0.0.1-SNAPSHOT.jar.original
                2 File(s)      14,493,271 bytes
                2 Dir(s)   307,447,808,000 bytes free

```

Now, run the JAR file by using the command **java -jar <JARFILE>**. Observe that in the above example, the JAR file is named **demo-0.0.1-SNAPSHOT.jar**

```

Command Prompt

C:\demo\target>java -jar demo-0.0.1-SNAPSHOT.jar

```

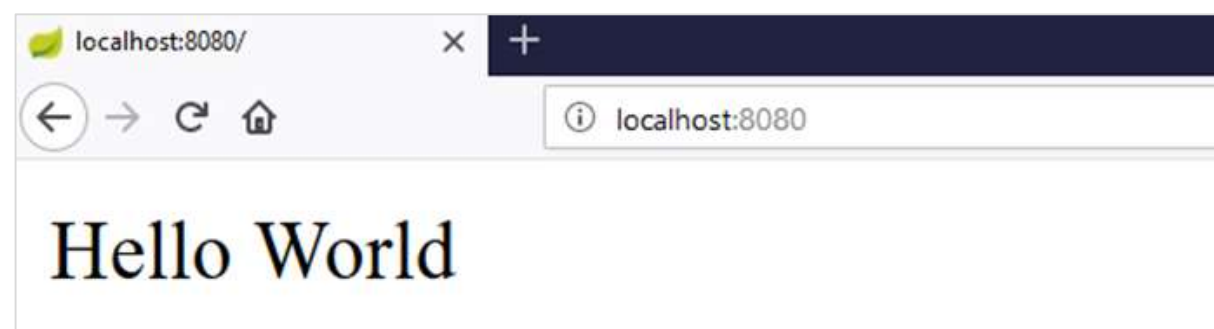
Once you run the jar file, you can see the output in the console window as shown below:

```

2017-12-02 12:28:32.244 INFO 6016 --- [main] o.s.j.e.s.AnnotationMethodExporter : Registering beans for JMX exposure on startup
2017-12-02 12:28:32.398 INFO 6016 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2017-12-02 12:28:32.411 INFO 6016 --- [main] com.tutorialspoint.demo.DemoApplication : Started DemoApplication in 6.902 seconds (JVM running for 8.119)

```

Now, look at the console, Tomcat started on port 8080 (http). Now, go to the web browser and hit the URL <http://localhost:8080/> and you can see the output as shown below:





# 4. Spring Boot – Tomcat Deployment

By using Spring Boot application, we can create a war file to deploy into the web server. In this chapter, you are going to learn how to create a WAR file and deploy the Spring Boot application in Tomcat web server.

## Spring Boot Servlet Initializer

The traditional way of deployment is making the Spring Boot Application **@SpringBootApplication** class extend the **SpringBootServletInitializer** class. Spring Boot Servlet Initializer class file allows you to configure the application when it is launched by using Servlet Container.

The code for Spring Boot Application class file for JAR file deployment is given below:

```
package com.tutorialspoint.demo;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

We need to extend the class **SpringBootServletInitializer** to support WAR file deployment. The code of Spring Boot Application class file is given below:

```
package com.tutorialspoint.demo;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.boot.web.support.SpringBootServletInitializer;

@SpringBootApplication
public class DemoApplication extends SpringBootServletInitializer {
    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
        return application.sources(DemoApplication.class);
    }
}
```

```

    }

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

```

## Setting Main Class

In Spring Boot, we need to mention the main class that should start in the build file. For this purpose, you can use the following pieces of code:

For Maven, add the start class in **pom.xml** properties as shown below:

```
<start-class>com.tutorialspoint.demo.DemoApplication</start-class>
```

For Gradle, add the main class name in **build.gradle** as shown below:

```
mainClassName="com.tutorialspoint.demo.DemoApplication"
```

## Update packaging JAR into WAR

We have to update the packaging JAR into WAR using the following pieces of code:

For Maven, add the packaging as WAR in **pom.xml** as shown below:

```
<packaging>war</packaging>
```

For Gradle, add the application plugin and war plugin in the **build.gradle** as shown below:

```

apply plugin: 'war'
apply plugin: 'application'

```

Now, let us write a simple Rest Endpoint to return the string "Hello World from Tomcat". To write a Rest Endpoint, we need to add the Spring Boot web starter dependency into our build file.

For Maven, add the Spring Boot starter dependency in **pom.xml** using the code as shown below:

```

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

```

For Gradle, add the Spring Boot starter dependency in **build.gradle** using the code as shown below:

```
dependencies {  
    compile('org.springframework.boot:spring-boot-starter-web')  
}
```

Now, write a simple Rest Endpoint in Spring Boot Application class file using the code as shown below:

```
package com.tutorialspoint.demo;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.boot.builder.SpringApplicationBuilder;  
import org.springframework.boot.web.support.SpringBootServletInitializer;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
@SpringBootApplication  
@RestController  
public class DemoApplication extends SpringBootServletInitializer {  
  
    @Override  
    protected SpringApplicationBuilder configure(SpringApplicationBuilder  
application) {  
        return application.sources(DemoApplication.class);  
    }  
  
    public static void main(String[] args) {  
        SpringApplication.run(DemoApplication.class, args);  
    }  
  
    @RequestMapping(value="/")  
    public String hello() {  
        return "Hello World from Tomcat";  
    }  
}
```

## Packaging your Application

Now, create a WAR file to deploy into the Tomcat server by using Maven and Gradle commands for packaging your application as given below:

For Maven, use the command **mvn package** for packaging your application. Then, the WAR file will be created and you can find it in the target directory as shown in the screenshots given below:

```

C:\demo>mvn package
  
```

```

C:\demo\target>dir
Volume in drive C has no label.
Volume Serial Number is 8CDD-0B1B

Directory of C:\demo\target

11/25/2017  12:56 PM    <DIR>          .
11/25/2017  12:56 PM    <DIR>          ..
11/25/2017  12:23 PM    <DIR>          classes
11/25/2017  12:56 PM    <DIR>          demo-0.0.1-SNAPSHOT
11/25/2017  12:24 PM             14,492,975 demo-0.0.1-SNAPSHOT.jar
11/25/2017  12:24 PM              3,078 demo-0.0.1-SNAPSHOT.jar.original
11/25/2017  12:56 PM             14,492,975 demo-0.0.1-SNAPSHOT.war
11/25/2017  12:56 PM             12,857,180 demo-0.0.1-SNAPSHOT.war.original
11/25/2017  12:23 PM    <DIR>          generated-sources
11/25/2017  12:23 PM    <DIR>          generated-test-sources
11/25/2017  12:55 PM    <DIR>          m2e-wtp
11/25/2017  12:24 PM    <DIR>          maven-archiver
11/25/2017  12:23 PM    <DIR>          maven-status
11/25/2017  12:24 PM    <DIR>          surefire-reports
11/25/2017  12:23 PM    <DIR>          test-classes
               4 File(s)      41,846,208 bytes
              11 Dir(s)  307,216,683,008 bytes free
  
```

For Gradle, use the command **gradle clean build** for packaging your application. Then, your WAR file will be created and you can find it under **build/libs** directory. Observe the screenshots given here for a better understanding:

```

C:\demo>gradle clean build
  
```

```

C:\demo\build\libs>dir
Volume in drive C has no label.
Volume Serial Number is 8CDD-0B1B

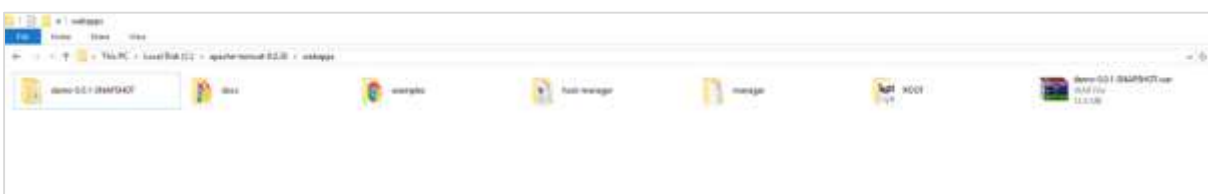
Directory of C:\demo\build\libs

11/25/2017  12:58 PM    <DIR>          .
11/25/2017  12:58 PM    <DIR>          ..
11/25/2017  12:58 PM             14,491,841 demo-0.0.1-SNAPSHOT.jar
11/25/2017  12:58 PM              1,724 demo-0.0.1-SNAPSHOT.jar.original
11/25/2017  12:58 PM             14,491,584 demo-0.0.1-SNAPSHOT.war
11/25/2017  12:58 PM             12,855,612 demo-0.0.1-SNAPSHOT.war.original
               4 File(s)      41,840,761 bytes
               2 Dir(s)   307,153,776,640 bytes free

```

## Deploy into Tomcat

Now, run the Tomcat Server, and deploy the WAR file under the **webapps** directory. Observe the screenshots shown here for a better understanding:



After successful deployment, hit the URL in your web browser <http://localhost:8080/demo-0.0.1-SNAPSHOT/> and observe that the output will look as shown in the screenshot given below:



The full code for this purpose is given below.

### **pom.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>com.tutorialspoint</groupId>
  <artifactId>demo</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>war</packaging>
  <name>demo</name>
  <description>Demo project for Spring Boot</description>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.5.8.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
    <java.version>1.8</java.version>
```

```

        <start-class>com.tutorialspoint.demo.DemoApplication</start-class>
    </properties>

    <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>

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

</project>

```

### build.gradle

```

buildscript {
    ext {
        springBootVersion = '1.5.8.RELEASE'
    }
    repositories {
        mavenCentral()
    }
    dependencies {

```

```

        classpath("org.springframework.boot:spring-boot-gradle-
plugin:${springBootVersion}")
    }
}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'org.springframework.boot'
apply plugin: 'war'
apply plugin: 'application'

group = 'com.tutorialspoint'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = 1.8
mainClassName="com.tutorialspoint.demo.DemoApplication"
repositories {
    mavenCentral()
}

dependencies {
    compile('org.springframework.boot:spring-boot-starter-web')
    testCompile('org.springframework.boot:spring-boot-starter-test')
}

```

The code for main Spring Boot application class file is given below:

```

package com.tutorialspoint.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.boot.web.support.SpringBootServletInitializer;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
@RestController
public class DemoApplication extends SpringBootServletInitializer {

```



```
@Override
protected SpringApplicationBuilder configure(SpringApplicationBuilder
application) {
    return application.sources(DemoApplication.class);
}

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

@RequestMapping(value="/")
public String hello() {
    return "Hello World from Tomcat";
}
}
```

# 5. Spring Boot – Build Systems

In Spring Boot, choosing a build system is an important task. We recommend Maven or Gradle as they provide a good support for dependency management. Spring does not support well other build systems.

## Dependency Management

---

Spring Boot team provides a list of dependencies to support the Spring Boot version for its every release. You do not need to provide a version for dependencies in the build configuration file. Spring Boot automatically configures the dependencies version based on the release. Remember that when you upgrade the Spring Boot version, dependencies also will upgrade automatically.

**Note:** If you want to specify the version for dependency, you can specify it in your configuration file. However, the Spring Boot team highly recommends that it is not needed to specify the version for dependency.

## Maven Dependency

---

For Maven configuration, we should inherit the Spring Boot Starter parent project to manage the Spring Boot Starters dependencies. For this, simply we can inherit the starter parent in our **pom.xml** file as shown below.

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.5.8.RELEASE</version>
</parent>
```

We should specify the version number for Spring Boot Parent Starter dependency. Then for other starter dependencies, we do not need to specify the Spring Boot version number. Observe the code given below:

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

## Gradle Dependency

---

We can import the Spring Boot Starters dependencies directly into **build.gradle** file. We do not need Spring Boot start Parent dependency like Maven for Gradle. Observe the code given below:

```
buildscript {  
    ext {  
        springBootVersion = '1.5.8.RELEASE'  
    }  
    repositories {  
        mavenCentral()  
    }  
    dependencies {  
        classpath("org.springframework.boot:spring-boot-gradle-  
plugin:${springBootVersion}")  
    }  
}
```

Similarly, in Gradle, we need not specify the Spring Boot version number for dependencies. Spring Boot automatically configures the dependency based on the version.

```
dependencies {  
    compile('org.springframework.boot:spring-boot-starter-web')  
}
```

# 6. Spring Boot – Code Structure

Spring Boot does not have any code layout to work with. However, there are some best practices that will help us. This chapter talks about them in detail.

## Default package

A class that does not have any package declaration is considered as a **default package**. Note that generally a default package declaration is not recommended. Spring Boot will cause issues such as malfunctioning of Auto Configuration or Component Scan, when you use default package.

**Note:** Java's recommended naming convention for package declaration is reversed domain name. For example: **com.tutorialspoint.myproject**

## Typical Layout

The typical layout of Spring Boot application is shown in the image given below:

```
com
+- tutorialspoint
    +- myproject
        +- Application.java
        |
        +- model
        | +- Product.java
        +- dao
        | +- ProductRepository.java
        +- controller
        | +- ProductController.java
        +- service
        | +- ProductService.java
```

The Application.java file should declare the main method along with @SpringBootApplication. Observe the code given below for a better understanding:

```
package com.tutorialspoint.myproject;
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);    }}
}
```

## 7. Spring Boot – Spring Beans and Dependency Injection

In Spring Boot, we can use Spring Framework to define our beans and their dependency injection. The **@ComponentScan** annotation is used to find beans and the corresponding injected with **@Autowired** annotation.

If you followed the Spring Boot typical layout, no need to specify any arguments for **@ComponentScan** annotation. All component class files are automatically registered with Spring Beans.

The following example provides an idea about Auto wiring the Rest Template object and creating a Bean for the same:

```
@Bean
public RestTemplate getRestTemplate() {
    return new RestTemplate();
}
```

The following code shows the code for auto wired Rest Template object and Bean creation object in main Spring Boot Application class file:

```
package com.tutorialspoint.demo;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.web.client.RestTemplate;
@SpringBootApplication
public class DemoApplication {
    @Autowired
    RestTemplate restTemplate;
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
    @Bean
    public RestTemplate getRestTemplate() {
        return new RestTemplate();
    }
}
```

## 8. Spring Boot – Runners

Application Runner and Command Line Runner interfaces lets you to execute the code after the Spring Boot application is started. You can use these interfaces to perform any actions immediately after the application has started. This chapter talks about them in detail.

### Application Runner

Application Runner is an interface used to execute the code after the Spring Boot application started. The example given below shows how to implement the Application Runner interface on the main class file.

```
package com.tutorialspoint.demo;
import org.springframework.boot.ApplicationArguments;
import org.springframework.boot.ApplicationRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class DemoApplication implements ApplicationRunner {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
    @Override
    public void run(ApplicationArguments arg0) throws Exception {
        System.out.println("Hello World from Application Runner");
    }
}
```

Now, if you observe the console window below **Hello World from Application Runner**, the println statement is executed after the Tomcat started. Is the following screenshot relevant?

```
2017-11-25 19:16:16.812 INFO 12696 --- [main] o.s.j.e.a.AnnotationMBeanExporter : Registering beans for JMX exposure on startup
2017-11-25 19:16:16.920 INFO 12696 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
Hello World from Application Runner
2017-11-25 19:16:16.926 INFO 12696 --- [main] com.tutorialspoint.demo.DemoApplication : Started DemoApplication in 4.161 seconds (JVM running for 4.864)
```

## Command Line Runner

Command Line Runner is an interface. It is used to execute the code after the Spring Boot application started. The example given below shows how to implement the Command Line Runner interface on the main class file.

```
package com.tutorialspoint.demo;

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

@SpringBootApplication
public class DemoApplication implements CommandLineRunner {

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

    @Override
    public void run(String... args) throws Exception {
        System.out.println("Hello world from Command Line Runner");
    }

}
```

Look at the console window below "Hello world from Command Line Runner" println statement is executed after the Tomcat started.

```
2017-11-25 19:22:33.316 INFO 12460 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
Hello world from Command Line Runner
2017-11-25 19:22:33.321 INFO 12460 --- [main] com.tutorialspoint.demo.DemoApplication : Started DemoApplication in 3.69 seconds (JVM running for 4.083)
```

# 9. Spring Boot – Application Properties

Application Properties support us to work in different environments. In this chapter, you are going to learn how to configure and specify the properties to a Spring Boot application.

## Command Line Properties

---

Spring Boot application converts the command line properties into Spring Boot Environment properties. Command line properties take precedence over the other property sources. By default, Spring Boot uses the 8080 port number to start the Tomcat. Let us learn how change the port number by using command line properties.

**Step 1:** After creating an executable JAR file, run it by using the command **java -jar <JARFILE>**.

**Step 2:** Use the command given in the screenshot given below to change the port number for Spring Boot application by using command line properties.



```
C:\demo\target>java -jar demo-0.0.1-SNAPSHOT.jar --server.port=9090
```

**Note:** You can provide more than one application properties by using the delimiter --.

## Properties File

---

Properties files are used to keep 'N' number of properties in a single file to run the application in a different environment. In Spring Boot, properties are kept in the **application.properties** file under the classpath.

The application.properties file is located in the **src/main/resources** directory. The code for sample **application.properties** file is given below:

```
server.port=9090
spring.application.name=demoservice
```

Note that in the code shown above the Spring Boot application **demoservice** starts on the port 9090.

## YAML File

---

Spring Boot supports YAML based properties configurations to run the application. Instead of **application.properties**, we can use **application.yml** file. This YAML file also should be kept inside the classpath. The sample **application.yml** file is given below:

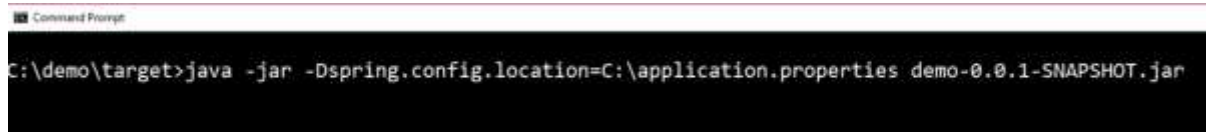


```
spring:
  application:
    name: demoservice
server:
  port: 9090
```

## Externalized Properties

Instead of keeping the properties file under classpath, we can keep the properties in different location or path. While running the JAR file, we can specify the properties file path. You can use the following command to specify the location of properties file while running the JAR:

```
-Dspring.config.location=C:\application.properties
```



```
C:\demo\target>java -jar -Dspring.config.location=C:\application.properties demo-0.0.1-SNAPSHOT.jar
```

## Use of @Value Annotation

The @Value annotation is used to read the environment or application property value in Java code. The syntax to read the property value is shown below:

```
@Value("${property_key_name}")
```

Look at the following example that shows the syntax to read the **spring.application.name** property value in Java variable by using @Value annotation.

```
@Value("${spring.application.name}")
```

Observe the code given below for a better understanding:

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
@RestController
public class DemoApplication {

    @Value("${spring.application.name}")
```

```

private String name;

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

@RequestMapping(value="/")
public String name() {
    return name;
}
}

```

**Note:** If the property is not found while running the application, Spring Boot throws the Illegal Argument exception as **Could not resolve placeholder 'spring.application.name' in value "\${spring.application.name}"**.

To resolve the placeholder issue, we can set the default value for the property using the syntax given below:

```

@Value("${property_key_name:default_value}")
@Value("${spring.application.name:demoservice}")

```

## Spring Boot Active Profile

Spring Boot supports different properties based on the Spring active profile. For example, we can keep two separate files for development and production to run the Spring Boot application.

### Spring active profile in application.properties

Let us understand how to have Spring active profile in application.properties. By default, application.properties will be used to run the Spring Boot application. If you want to use profile based properties, we can keep separate properties file for each profile as shown below:

#### application.properties

```

server.port=8080
spring.application.name=demoservice

```

#### application-dev.properties

```

server.port=9090
spring.application.name=demoservice

```

**application-prod.properties**

```
server.port=4431
spring.application.name=demoservice
```

While running the JAR file, we need to specify the spring active profile based on each properties file. By default, Spring Boot application uses the application.properties file. The command to set the spring active profile is shown below:

```
Command Prompt
C:\demo\target>java -jar demo-0.0.1-SNAPSHOT.jar --spring.profiles.active=dev
```

You can see active profile name on the console log as shown below:

```
2017-11-26 08:13:16.322 INFO 14028 --- [           main]
com.tutorialspoint.demo.DemoApplication : The following profiles are active:
dev
```

Now, Tomcat has started on the port 9090 (http) as shown below:

```
2017-11-26 08:13:20.185 INFO 14028 --- [           main]
s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 9090
(http)
```

You can set the Production active profile as shown below:

```
Command Prompt
C:\demo\target>java -jar demo-0.0.1-SNAPSHOT.jar --spring.profiles.active=prod
```

You can see active profile name on the console log as shown below:

```
2017-11-26 08:13:16.322 INFO 14028 --- [           main]
com.tutorialspoint.demo.DemoApplication : The following profiles are active:
prod
```

Now, Tomcat started on the port 4431 (http) as shown below:

```
2017-11-26 08:13:20.185 INFO 14028 --- [           main]
s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 4431
(http)
```

**Spring active profile for application.yml**

Let us understand how to keep Spring active profile for application.yml. We can keep the Spring active profile properties in the single **application.yml** file. No need to use the separate file like application.properties.

The following is an example code to keep the Spring active profiles in application.yml file. Note that the delimiter (---) is used to separate each profile in application.yml file.

```

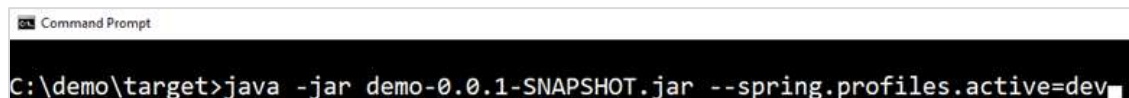
spring:
  application:
    name: demoservice
server:
  port: 8080

---
spring:
  profiles: dev
  application:
    name: demoservice
server:
  port: 9090

---
spring:
  profiles: prod
  application:
    name: demoservice
server:
  port: 4431

```

To command to set development active profile is given below:



```

C:\demo\target>java -jar demo-0.0.1-SNAPSHOT.jar --spring.profiles.active=dev

```

You can see active profile name on the console log as shown below:

```

2017-11-26 08:41:37.202 INFO 14104 --- [           main]
com.tutorialspoint.demo.DemoApplication : The following profiles are active:
dev

```

Now, Tomcat started on the port 9090 (http) as shown below:

```

2017-11-26 08:41:46.650 INFO 14104 --- [           main]
s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 9090
(http)

```

The command to set Production active profile is given below:

```
Command Prompt
C:\demo\target>java -jar demo-0.0.1-SNAPSHOT.jar --spring.profiles.active=prod
```

You can see active profile name on the console log as shown below:

```
2017-11-26 08:43:10.743 INFO 13400 --- [          main]
com.tutorialspoint.demo.DemoApplication : The following profiles are active:
prod
```

This will start Tomcat on the port 4431 (http) as shown below:

```
2017-11-26 08:43:14.473 INFO 13400 --- [          main]
s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 4431
(http)
```

# 10. Spring Boot – Logging

Spring Boot uses Apache Commons logging for all internal logging. Spring Boot's default configurations provides a support for the use of Java Util Logging, Log4j2, and Logback. Using these, we can configure the console logging as well as file logging.

If you are using Spring Boot Starters, Logback will provide a good support for logging. Besides, Logback also provides a use of good support for Common Logging, Util Logging, Log4J, and SLF4J.

## Log Format

The default Spring Boot Log format is shown in the screenshot given below.

```
2017-11-26 09:30:27.873 INFO 5040 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat initialized with port(s): 8080 (http)
2017-11-26 09:30:27.895 INFO 5040 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2017-11-26 09:30:27.898 INFO 5040 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet Engine: Apache Tomcat/8.5.23
2017-11-26 09:30:28.040 INFO 5040 --- [ost-startStop-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2017-11-26 09:30:28.040 INFO 5040 --- [ost-startStop-1] o.s.web.context.ContextLoader : Root WebApplicationContext: initialization completed in 2759 ms
```

which gives you the following information:

- **Date** and **Time** that gives the date and time of the log
- **Log level** shows INFO, ERROR or WARN
- **Process ID**
- The --- which is a separator
- **Thread name** is enclosed within the square brackets []
- **Logger Name** that shows the Source class name
- The Log message

## Console Log Output

The default log messages will print to the console window. By default, "INFO", "ERROR" and "WARN" log messages will print in the log file.

If you have to enable the debug level log, add the debug flag on starting your application using the command shown below:

```
java -jar demo.jar --debug
```

You can also add the debug mode to your application.properties file as shown here:

```
debug=true
```

## File Log Output

By default, all logs will print on the console window and not in the files. If you want to print the logs in a file, you need to set the property **logging.file** or **logging.path** in the application.properties file.

You can specify the log file path using the property shown below. Note that the log file name is spring.log.

```
logging.path=/var/tmp/
```

You can specify the own log file name using the property shown below:

```
logging.file=/var/tmp/mylog.log
```

**Note:** Log files will rotate automatically after reaching the size 10 MB.

## Log Levels

Spring Boot supports all logger levels such as "TRACE", "DEBUG", "INFO", "WARN", "ERROR", "FATAL", "OFF". You can define Root logger in the application.properties file as shown below:

```
logging.level.root=WARN
```

**Note:** Logback does not support "FATAL" level log. It is mapped to the "ERROR" level log.

## Configure Logback

Logback supports XML based configuration to handle Spring Boot Log configurations. Logging configuration details are configured in **logback.xml** file. The logback.xml file should be placed under the classpath.

You can configure the ROOT level log in Logback.xml file using the code given below:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <root level="INFO">
    </root>
</configuration>
```

You can configure the console appender in Logback.xml file given below.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    </appender>
    <root level="INFO">
        <appender-ref ref="STDOUT"/>
    </root>
</configuration>
```

You can configure the file appender in Logback.xml file using the code given below. Note that you need to specify the Log file path insider the file appender.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <appender name="FILE" class="ch.qos.logback.core.FileAppender">
        <File>/var/tmp/mylog.log</File>
    </appender>
<root level="INFO">
    <appender-ref ref="FILE"/>
</root>
</configuration>
```

You can define the Log pattern in **logback.xml** file using the code given below. You can also define the set of supported log patterns inside the console or file log appender using the code given below:

```
<pattern>[%d{yyyy-MM-dd'T'HH:mm:ss.sss'Z'}} [%C] [%t] [%L] [%-5p]
%m%n</pattern>
```

The code for complete logback.xml file is given below. You have to place this in the class path.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>[%d{yyyy-MM-dd'T'HH:mm:ss.sss'Z'}} [%C] [%t] [%L] [%-5p]
            %m%n</pattern>
        </encoder>
    </appender>
    <appender name="FILE" class="ch.qos.logback.core.FileAppender">
        <File>/var/tmp/mylog.log</File>
        <encoder>
            <pattern>[%d{yyyy-MM-dd'T'HH:mm:ss.sss'Z'}} [%C] [%t] [%L] [%-5p]
            %m%n</pattern>
        </encoder>
    </appender>
    <root level="INFO">
        <appender-ref ref="FILE"/>
        <appender-ref ref="STDOUT"/>
    </root>
</configuration>
```



The code given below shows how to add the slf4j logger in Spring Boot main class file.

```
package com.tutorialspoint.demo;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class DemoApplication {
    private static final Logger logger =
    LoggerFactory.getLogger(DemoApplication.class);
    public static void main(String[] args) {
        logger.info("this is a info message");
        logger.warn("this is a warn message");
        logger.error("this is a error message");
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

The output that you can see in the console window is shown here:

```
[2017-11-26T11:38:07.007Z] [com.tutorialspoint.demo.DemoApplication] [main] [14] [INFO ] this is a info message
[2017-11-26T11:38:07.007Z] [com.tutorialspoint.demo.DemoApplication] [main] [15] [WARN ] this is a warn message
[2017-11-26T11:38:07.007Z] [com.tutorialspoint.demo.DemoApplication] [main] [16] [ERROR] this is a error message
```

The output that you can see in the log file is shown here:

```
[2017-11-26T11:38:07.007Z] [com.tutorialspoint.demo.DemoApplication] [main] [14] [INFO ] this is a info message
[2017-11-26T11:38:07.007Z] [com.tutorialspoint.demo.DemoApplication] [main] [15] [WARN ] this is a warn message
[2017-11-26T11:38:07.007Z] [com.tutorialspoint.demo.DemoApplication] [main] [16] [ERROR] this is a error message
```

# 11. Spring Boot – Building RESTful Web Services

Spring Boot provides a very good support to building RESTful Web Services for enterprise applications. This chapter will explain in detail about building RESTful web services using Spring Boot.

**Note:** For building a RESTful Web Services, we need to add the Spring Boot Starter Web dependency into the build configuration file.

If you are a Maven user, use the following code to add the below dependency in your **pom.xml** file:

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

If you are a Gradle user, use the following code to add the below dependency in your **build.gradle** file.

```
compile('org.springframework.boot:spring-boot-starter-web')
```

The code for complete build configuration file **Maven build – pom.xml** is given below:

```
<?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>com.tutorialspoint</groupId>
    <artifactId>demo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>
    <name>demo</name>
    <description>Demo project for Spring Boot</description>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.8.RELEASE</version>
        <relativePath/>
```

```

</parent>
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
    <java.version>1.8</java.version>
</properties>
<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>
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
</project>

```

The code for complete build configuration file **Gradle Build – build.gradle** is given below:

```

buildscript {
    ext {
        springBootVersion = '1.5.8.RELEASE'
    }
    repositories {
        mavenCentral()
    }
    dependencies {

```

```

        classpath("org.springframework.boot:spring-boot-gradle-
plugin:${springBootVersion}")
    }
}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'org.springframework.boot'

group = 'com.tutorialspoint'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = 1.8

repositories {
    mavenCentral()
}

dependencies {
    compile('org.springframework.boot:spring-boot-starter-web')
    testCompile('org.springframework.boot:spring-boot-starter-test')
}

```

Before you proceed to build a RESTful web service, it is suggested that you have knowledge of the following annotations:

## Rest Controller

The `@RestController` annotation is used to define the RESTful web services. It serves JSON, XML and custom response. Its syntax is shown below:

```

@RestController
public class ProductServiceController {
}

```

## Request Mapping

The `@RequestMapping` annotation is used to define the Request URI to access the REST Endpoints. We can define Request method to consume and produce object. The default request method is GET.

```
@RequestMapping(value="/products")  
public ResponseEntity<Object> getProducts() { }
```

## Request Body

The @RequestBody annotation is used to define the request body content type.

```
public ResponseEntity<Object> createProduct(@RequestBody Product product) {  
}
```

## Path Variable

The @PathVariable annotation is used to define the custom or dynamic request URI. The Path variable in request URI is defined as curly braces {} as shown below:

```
public ResponseEntity<Object> updateProduct(@PathVariable("id") String id) {  
}
```

## Request Parameter

The @RequestParam annotation is used to read the request parameters from the Request URL. By default, it is a required parameter. We can also set default value for request parameters as shown here:

```
public ResponseEntity<Object> getProduct(@RequestParam(value="name",  
required=false, defaultValue="honey") String name) {  
}
```

## GET API

The default HTTP request method is GET. This method does not require any Request Body. You can send request parameters and path variables to define the custom or dynamic URL.

The sample code to define the HTTP GET request method is shown below. In this example, we used HashMap to store the Product. Note that we used a POJO class as the product to be stored.

Here, the request URI is **/products** and it will return the list of products from HashMap repository. The controller class file is given below that contains GET method REST Endpoint.

```
package com.tutorialspoint.demo.controller;

import java.util.HashMap;
import java.util.Map;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.tutorialspoint.demo.model.Product;

@RestController
public class ProductServiceController {

    private static Map<String, Product> productRepo = new HashMap<>();

    static {

        Product honey = new Product();
        honey.setId("1");
        honey.setName("Honey");
        productRepo.put(honey.getId(), honey);

        Product almond = new Product();
        almond.setId("2");
        almond.setName("Almond");
        productRepo.put(almond.getId(), almond);

    }

    @RequestMapping(value="/products")
    public ResponseEntity<Object> getProduct() {
        return new ResponseEntity<>(productRepo.values(), HttpStatus.OK);
    }
}
```

```
}
}
```

## POST API

The HTTP POST request is used to create a resource. This method contains the Request Body. We can send request parameters and path variables to define the custom or dynamic URL.

The following example shows the sample code to define the HTTP POST request method. In this example, we used HashMap to store the Product, where the product is a POJO class.

Here, the request URI is **/products**, and it will return the String after storing the product into HashMap repository.

```
package com.tutorialspoint.demo.controller;

import java.util.HashMap;
import java.util.Map;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

import com.tutorialspoint.demo.model.Product;

@RestController
public class ProductServiceController {
    private static Map<String, Product> productRepo = new HashMap<>();

    @RequestMapping(value="/products", method=RequestMethod.POST)
    public ResponseEntity<Object> createProduct(@RequestBody Product product)
    {
        productRepo.put(product.getId(), product);
        return new ResponseEntity<>("Product is created successfully",
        HttpStatus.CREATED);
    }
}
```

## PUT API

The HTTP PUT request is used to update the existing resource. This method contains a Request Body. We can send request parameters and path variables to define the custom or dynamic URL.

The example given below shows how to define the HTTP PUT request method. In this example, we used HashMap to update the existing Product, where the product is a POJO class.

Here the request URI is **/products/{id}**, which will return the String after a the product into a HashMap repository. Note that we used the Path variable **{id}** which defines the products ID that needs to be updated.

```
package com.tutorialspoint.demo.controller;

import java.util.HashMap;
import java.util.Map;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;
import com.tutorialspoint.demo.model.Product;

@RestController
public class ProductServiceController {

    private static Map<String, Product> productRepo = new HashMap<>();

    @RequestMapping(value="/products/{id}", method=RequestMethod.PUT)
    public ResponseEntity<Object> updateProduct(@PathVariable("id") String id,
        @RequestBody Product product) {
        productRepo.remove(id);
        product.setId(id);
        productRepo.put(id, product);
        return new ResponseEntity<>("Product is updated successssfully",
            HttpStatus.OK);
    }
}
```



## DELETE API

The HTTP Delete request is used to delete the existing resource. This method does not contain any Request Body. We can send request parameters and path variables to define the custom or dynamic URL.

The example given below shows how to define the HTTP DELETE request method. In this example, we used HashMap to remove the existing product, which is a POJO class.

The request URI is **/products/{id}** and it will return the String after deleting the product from HashMap repository. We used the Path variable **{id}** which defines the products ID that needs to be deleted.

```
package com.tutorialspoint.demo.controller;

import java.util.HashMap;
import java.util.Map;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

import com.tutorialspoint.demo.model.Product;

@RestController
public class ProductServiceController {

    private static Map<String, Product> productRepo = new HashMap<>();

    @RequestMapping(value="/products/{id}", method=RequestMethod.DELETE)
    public ResponseEntity<Object> delete(@PathVariable("id") String id) {
        productRepo.remove(id);
        return new ResponseEntity<>("Product is deleted successssfully",
        HttpStatus.OK);
    }
}
```

This section gives you the complete set of source code. Observe the following codes for their respective functionalities:

### The Spring Boot main application class – DemoApplication.java

```
package com.tutorialspoint.demo;

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

@SpringBootApplication
public class DemoApplication {

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

}
```

### The POJO class – Product.java

```
package com.tutorialspoint.demo.model;

public class Product {

    private String id;
    private String name;

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

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

}
```

### The Rest Controller class – ProductServiceController.java

```

package com.tutorialspoint.demo.controller;

import java.util.HashMap;
import java.util.Map;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

import com.tutorialspoint.demo.model.Product;

@RestController
public class ProductServiceController {

    private static Map<String, Product> productRepo = new HashMap<>();

    static {

        Product honey = new Product();
        honey.setId("1");
        honey.setName("Honey");
        productRepo.put(honey.getId(), honey);

        Product almond = new Product();
        almond.setId("2");
        almond.setName("Almond");
        productRepo.put(almond.getId(), almond);

    }

    @RequestMapping(value="/products/{id}", method=RequestMethod.DELETE)
    public ResponseEntity<Object> delete(@PathVariable("id") String id) {

```

```

        productRepo.remove(id);
        return new ResponseEntity<>("Product is deleted successssfully",
HttpStatus.OK);
    }

    @RequestMapping(value="/products/{id}", method=RequestMethod.PUT)
    public ResponseEntity<Object> updateProduct(@PathVariable("id") String id,
@RequestBody Product product) {
        productRepo.remove(id);
        product.setId(id);
        productRepo.put(id, product);
        return new ResponseEntity<>("Product is updated successssfully",
HttpStatus.OK);
    }

    @RequestMapping(value="/products", method=RequestMethod.POST)
    public ResponseEntity<Object> createProduct(@RequestBody Product product)
    {
        productRepo.put(product.getId(), product);
        return new ResponseEntity<>("Product is created successfully",
HttpStatus.CREATED);
    }

    @RequestMapping(value="/products")
    public ResponseEntity<Object> getProduct() {
        return new ResponseEntity<>(productRepo.values(), HttpStatus.OK);
    }
}

```

You can create an executable JAR file, and run the spring boot application by using the below Maven or Gradle commands as shown:

For Maven, use the command shown below:

```
mvn clean install
```

After "BUILD SUCCESS", you can find the JAR file under the target directory.

For Gradle, use the command shown below:

```
gradle clean build
```

After "BUILD SUCCESSFUL", you can find the JAR file under the build/libs directory.

You can run the JAR file by using the command shown below:

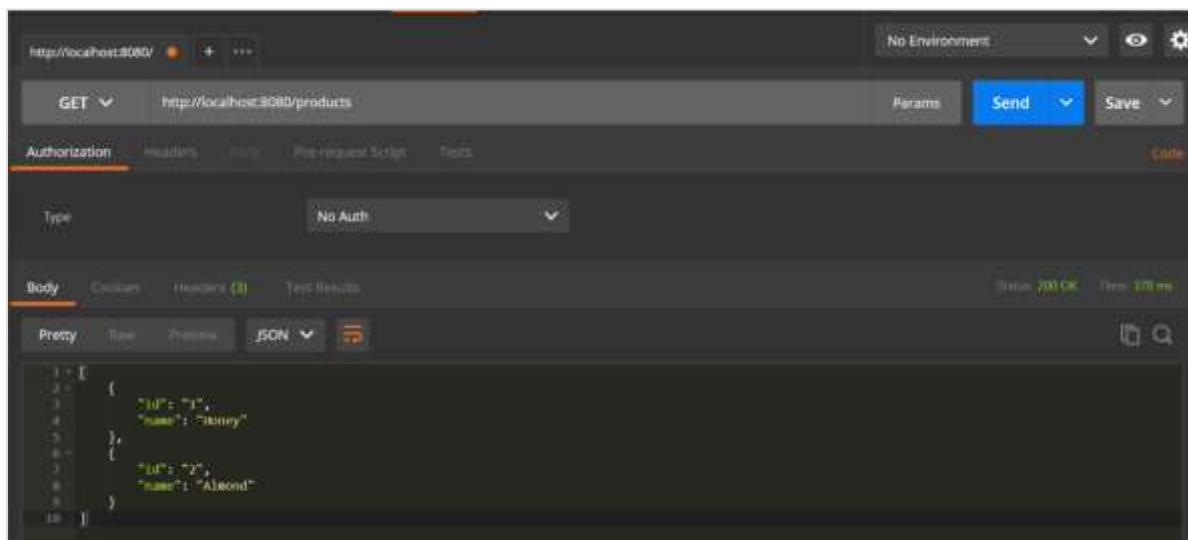
```
java -jar <JARFILE>
```

This will start the application on the Tomcat port 8080 as shown below:

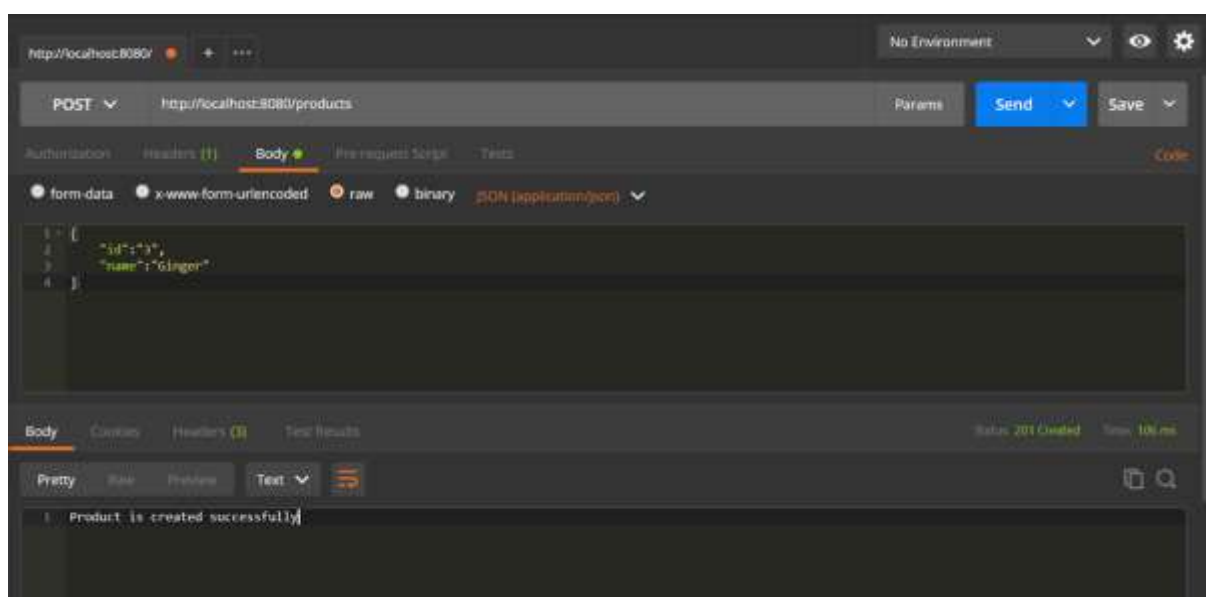
```
2017-11-26 13:56:27.912 INFO 13204 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2017-11-26 13:56:27.922 INFO 13204 --- [main] com.tutorialspoint.demo.DemoApplication : Started DemoApplication in 5.961 seconds (2017-11-26 13:56:27.922)
M running for 6.933)
```

Now hit the URL shown below in POSTMAN application and see the output.

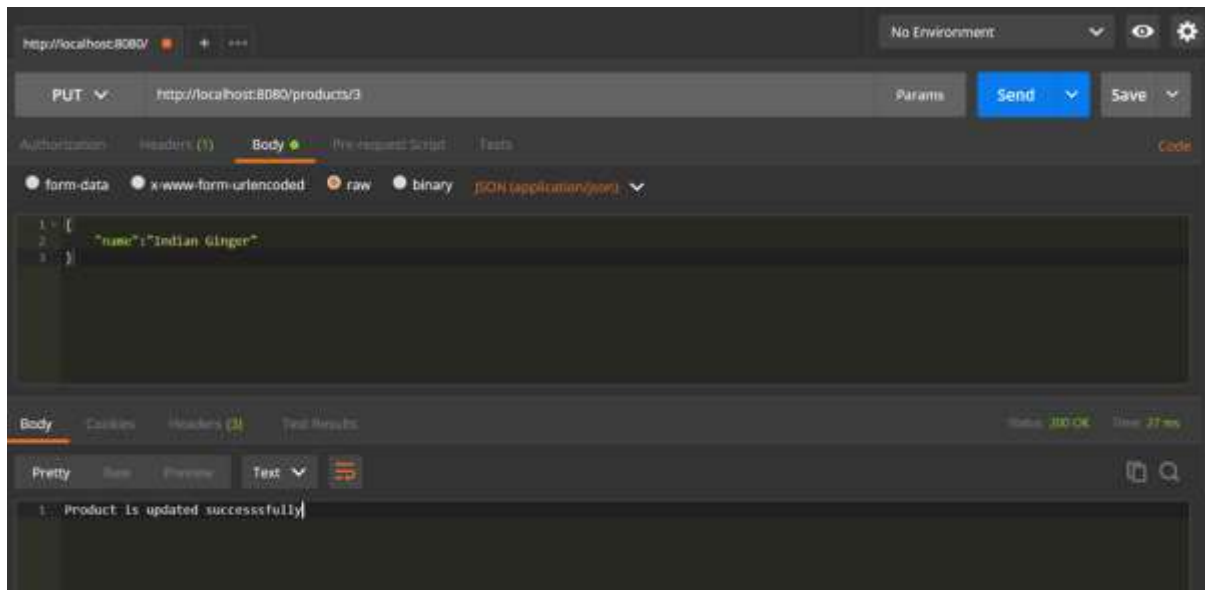
GET API URL is: <http://localhost:8080/products>



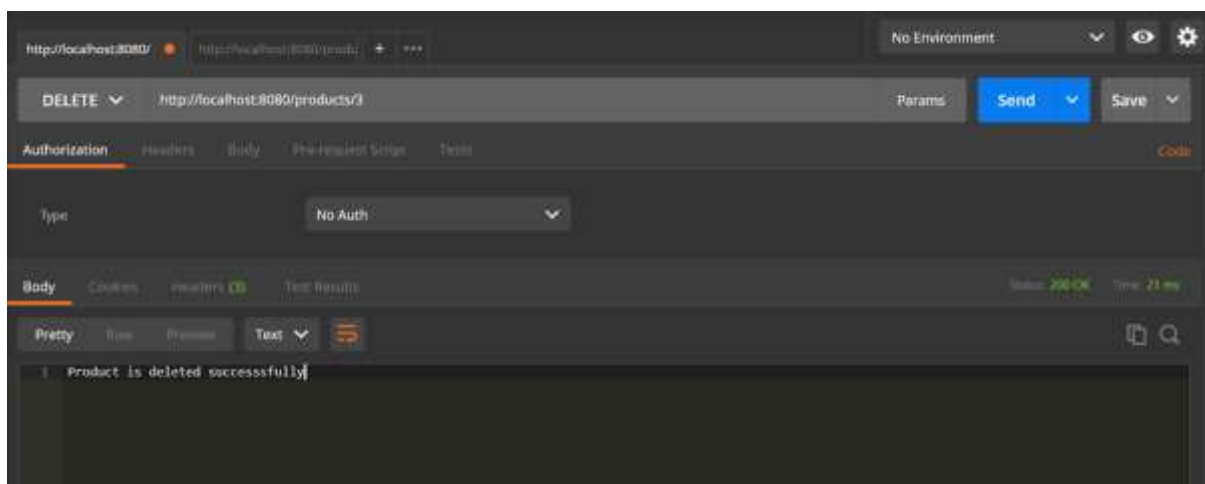
POST API URL is: <http://localhost:8080/products>



PUT API URL is: <http://localhost:8080/products/3>



DELETE API URL is: <http://localhost:8080/products/3>



# 12. Spring Boot – Exception Handling

Handling exceptions and errors in APIs and sending the proper response to the client is good for enterprise applications. In this chapter, we will learn how to handle exceptions in Spring Boot.

Before proceeding with exception handling, let us gain an understanding on the following annotations.

## Controller Advice

---

The @ControllerAdvice is an annotation, to handle the exceptions globally.

## Exception Handler

---

The @ExceptionHandler is an annotation used to handle the specific exceptions and sending the custom responses to the client.

You can use the following code to create @ControllerAdvice class to handle the exceptions globally:

```
package com.tutorialspoint.demo.exception;

import org.springframework.web.bind.annotation.ControllerAdvice;

@ControllerAdvice
public class ProductExceptionHandler {

}
```

Define a class that extends the RuntimeException class.

```
package com.tutorialspoint.demo.exception;

public class ProductNotFoundException extends RuntimeException {

    private static final long serialVersionUID = 1L;

}
```

You can define the `@ExceptionHandler` method to handle the exceptions as shown. This method should be used for writing the Controller Advice class file.

```
@ExceptionHandler(value = ProductNotFoundException.class)
public ResponseEntity<Object> exception(ProductNotFoundException exception) {
}
```

Now, use the code given below to throw the exception from the API.

```
@RequestMapping(value="/products/{id}", method=RequestMethod.PUT)
public ResponseEntity<Object> updateProduct() {
    throw new ProductNotFoundException();
}
```

The complete code to handle the exception is given below. In this example, we used the PUT API to update the product. Here, while updating the product, if the product is not found, then return the response error message as "Product not found". Note that the **ProductNotFoundException** exception class should extend the **RuntimeException**.

```
package com.tutorialspoint.demo.exception;

public class ProductNotFoundException extends RuntimeException {
    private static final long serialVersionUID = 1L;
}
```

The Controller Advice class to handle the exception globally is given below. We can define any Exception Handler methods in this class file.

```
package com.tutorialspoint.demo.exception;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;

@ControllerAdvice
public class ProductExceptionHandler {

    @ExceptionHandler(value = ProductNotFoundException.class)
    public ResponseEntity<Object> exception(ProductNotFoundException
exception) {
        return new ResponseEntity<>("Product not found",
HttpStatus.NOT_FOUND);
    }
}
```



The Product Service API controller file is given below to update the Product. If the Product is not found, then it throws the **ProductNotFoundException** class.

```
package com.tutorialspoint.demo.controller;

import java.util.HashMap;
import java.util.Map;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

import com.tutorialspoint.demo.exception.ProductNotFoundException;
import com.tutorialspoint.demo.model.Product;

@RestController
public class ProductServiceController {

    private static Map<String, Product> productRepo = new HashMap<>();

    static {

        Product honey = new Product();
        honey.setId("1");
        honey.setName("Honey");
        productRepo.put(honey.getId(), honey);

        Product almond = new Product();
        almond.setId("2");
        almond.setName("Almond");
        productRepo.put(almond.getId(), almond);

    }
}
```

```

    @RequestMapping(value="/products/{id}", method=RequestMethod.PUT)
    public ResponseEntity<Object> updateProduct(@PathVariable("id") String id,
    @RequestBody Product product) {
        if(!productRepo.containsKey(id))
            throw new ProductNotFoundException();
        productRepo.remove(id);
        product.setId(id);
        productRepo.put(id, product);
        return new ResponseEntity<>("Product is updated successfully",
    HttpStatus.OK);
    }
}

```

The code for main Spring Boot application class file is given below:

```

package com.tutorialspoint.demo;

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

@SpringBootApplication
public class DemoApplication {

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

}

```

The code for **POJO class** for Product is given below:

```

package com.tutorialspoint.demo.model;

public class Product {
    private String id;
    private String name;

    public String getId() {
        return id;
    }
}

```

```

    public void setId(String id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

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

```

The code for **Maven build – pom.xml** is shown below:

```

<?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>com.tutorialspoint</groupId>
    <artifactId>demo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>
    <name>demo</name>
    <description>Demo project for Spring Boot</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.8.RELEASE</version>
        <relativePath/>
    </parent>

    <properties>

```

```

        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
    </properties>

    <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>

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

</project>

```

The code for **Gradle Build – build.gradle** is given below:

```

buildscript {
    ext {
        springBootVersion = '1.5.8.RELEASE'
    }
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-
plugin:${springBootVersion}")
    }
}
apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'org.springframework.boot'
group = 'com.tutorialspoint'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = 1.8
repositories {
    mavenCentral()
}
dependencies {
    compile('org.springframework.boot:spring-boot-starter-web')
    testCompile('org.springframework.boot:spring-boot-starter-test')
}

```

You can create an executable JAR file, and run the spring boot application by using the following Maven or Gradle commands.

For Maven, you can use the following command:

```
mvn clean install
```

After "BUILD SUCCESS", you can find the JAR file under the target directory.

For Gradle, you can use the following command:

```
gradle clean build
```

After "BUILD SUCCESSFUL", you can find the JAR file under the build/libs directory.

You can run the JAR file by using the following command:

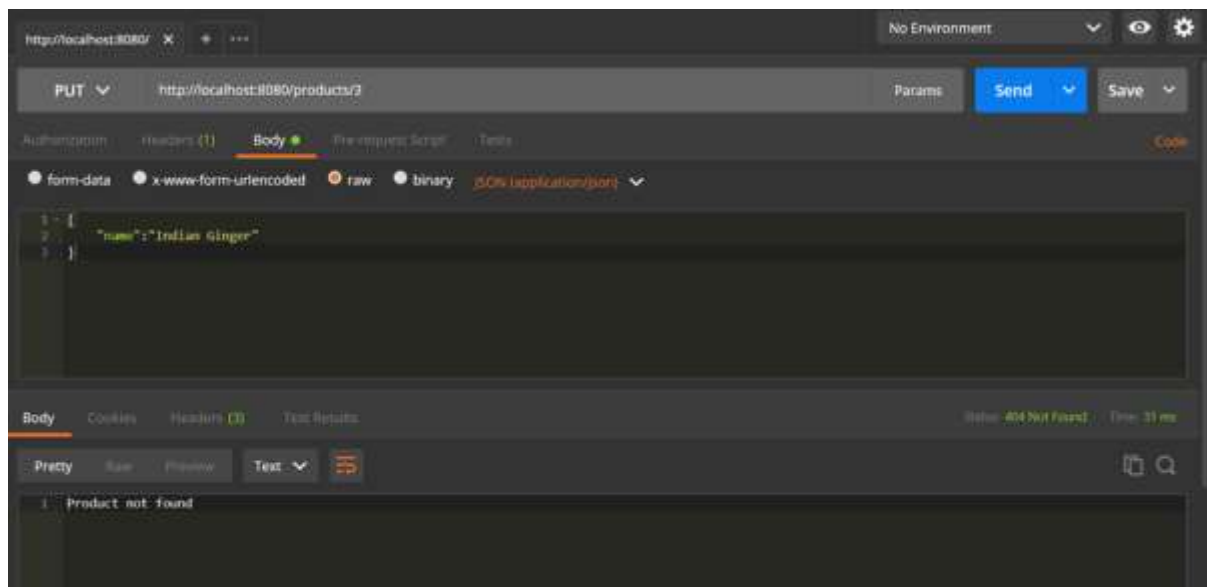
```
java -jar <JARFILE>
```

This will start the application on the Tomcat port 8080 as shown below:

```
2017-11-26 13:56:27.912 INFO 13204 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2017-11-26 13:56:27.922 INFO 13204 --- [main] com.tutorialspoint.demo.DemoApplication : Started DemoApplication in 5.961 seconds (30
M running for 6.933)
```

Now hit the below URL in POSTMAN application and you can see the output as shown below:

Update URL: <http://localhost:8080/products/3>



# 13. Spring Boot – Interceptor

You can use the Interceptor in Spring Boot to perform operations under the following situations:

- Before sending the request to the controller
- Before sending the response to the client

For example, you can use an interceptor to add the request header before sending the request to the controller and add the response header before sending the response to the client.

To work with interceptor, you need to create **@Component** class that supports it and it should implement the **HandlerInterceptor** interface.

The following are the three methods you should know about while working on Interceptors:

- **preHandle()** method: This is used to perform operations before sending the request to the controller. This method should return true to return the response to the client.
- **postHandle()** method: This is used to perform operations before sending the response to the client.
- **afterCompletion()** method: This is used to perform operations after completing the request and response.

Observe the following code for a better understanding:

```
@Component
public class ProductServiceInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {
        return true;
    }
    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, ModelAndView modelAndView) throws Exception {}
    @Override
    public void afterCompletion(HttpServletRequest request,
HttpServletResponse response, Object handler, Exception exception) throws
Exception {}
}
```

You will have to register this Interceptor with **InterceptorRegistry** by using **WebMvcConfigurerAdapter** as shown below:

```
@Component
public class ProductServiceInterceptorAppConfig extends WebMvcConfigurerAdapter
{

    @Autowired
    ProductServiceInterceptor productServiceInterceptor;

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(productServiceInterceptor);
    }
}
```

In the example given below, we are going to hit the GET products API which gives the output as given under:

The code for the Interceptor class ProductServiceInterceptor.java is given below:

```
package com.tutorialspoint.demo.interceptor;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.stereotype.Component;
import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;

@Component
public class ProductServiceInterceptor implements HandlerInterceptor {

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {
        System.out.println("Pre Handle method is Calling");
        return true;
    }

    @Override
```



```

        public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, ModelAndView modelAndView) throws Exception {
            System.out.println("Post Handle method is Calling");
        }

        @Override
        public void afterCompletion(HttpServletRequest request,
HttpServletResponse response, Object handler, Exception exception) throws
Exception {
            System.out.println("Request and Response is completed");
        }
    }
}

```

The code for Application Configuration class file to register the Interceptor into Interceptor Registry – ProductServiceInterceptorAppConfig.java is given below:

```

package com.tutorialspoint.demo.interceptor;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import
org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;

@Component
public class ProductServiceInterceptorAppConfig extends WebMvcConfigurerAdapter
{

    @Autowired
    ProductServiceInterceptor productServiceInterceptor;

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(productServiceInterceptor);
    }
}

```

The code for Controller class file ProductServiceController.java is given below:

```
package com.tutorialspoint.demo.controller;

import java.util.HashMap;
import java.util.Map;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;
import com.tutorialspoint.demo.exception.ProductNotFoundException;
import com.tutorialspoint.demo.model.Product;

@RestController
public class ProductServiceController {

    private static Map<String, Product> productRepo = new HashMap<>();

    static {

        Product honey = new Product();
        honey.setId("1");
        honey.setName("Honey");
        productRepo.put(honey.getId(), honey);

        Product almond = new Product();
        almond.setId("2");
        almond.setName("Almond");
        productRepo.put(almond.getId(), almond);

    }

    @RequestMapping(value="/products")
    public ResponseEntity<Object> getProduct() {

        return new ResponseEntity<>(productRepo.values(), HttpStatus.OK);

    }

}
```

The code for POJO class for Product.java is given below:

```
package com.tutorialspoint.demo.model;

public class Product {
    private String id;
    private String name;

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

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

The code for main Spring Boot application class file **DemoApplication.java** is given below:

```
package com.tutorialspoint.demo;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args); }
}
```

The code for Maven build – **pom.xml** is shown here:

```

<?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>com.tutorialspoint</groupId>
    <artifactId>demo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>
    <name>demo</name>
    <description>Demo project for Spring Boot</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.8.RELEASE</version>
        <relativePath/>
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
    </properties>

    <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>

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

</project>

```

The code for Gradle Build **build.gradle** is shown here:

```

buildscript {
    ext {
        springBootVersion = '1.5.8.RELEASE'
    }
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-
plugin:${springBootVersion}")
    }
}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'org.springframework.boot'
group = 'com.tutorialspoint'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = 1.8

repositories {

```

```

        mavenCentral()
    }
    dependencies {
        compile('org.springframework.boot:spring-boot-starter-web')
        testCompile('org.springframework.boot:spring-boot-starter-test')
    }

```

You can create an executable JAR file, and run the Spring Boot application by using the below Maven or Gradle commands.

For Maven, use the command as shown below:

```
mvn clean install
```

After "BUILD SUCCESS", you can find the JAR file under the target directory.

For Gradle, use the command as shown below:

```
gradle clean build
```

After "BUILD SUCCESSFUL", you can find the JAR file under the build/libs directory.

You can run the JAR file by using the following command:

```
java -jar <JARFILE>
```

Now, the application has started on the Tomcat port 8080 as shown below:

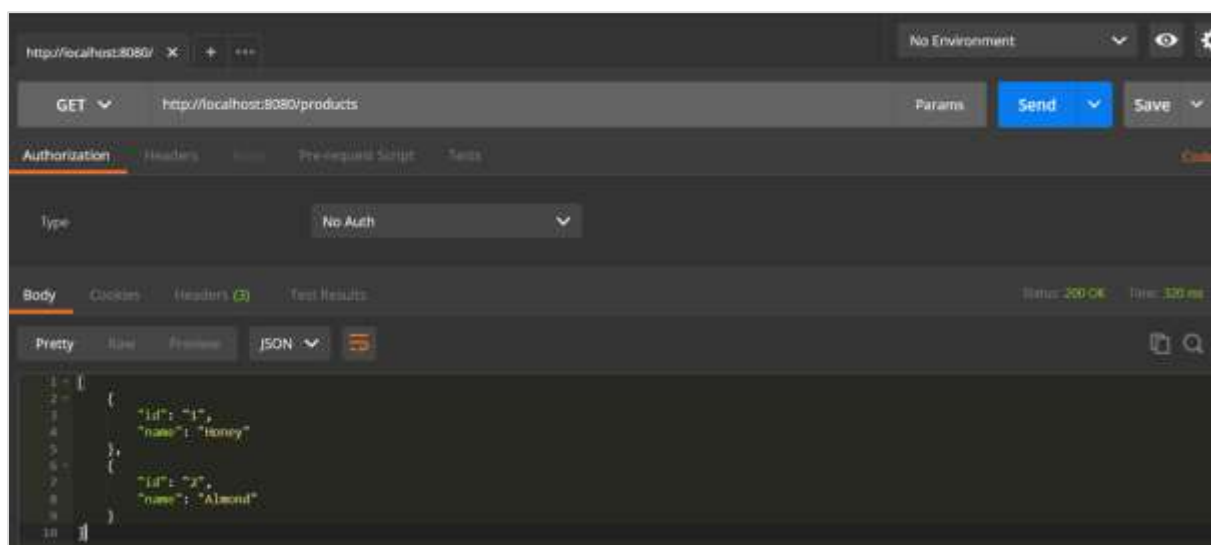
```

2017-11-26 13:56:27.912 INFO 13204 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2017-11-26 13:56:27.922 INFO 13204 --- [main] com.tutorialspoint.demo.DemoApplication : Started DemoApplication in 5.961 seconds (JVM running for 6.933)

```

Now hit the below URL in POSTMAN application and you can see the output as shown under:

GET API: <http://localhost:8080/products>



In the console window, you can see the `System.out.println` statements added in the `Interceptor` as shown in the screenshot given below:

```
Pre Handle method is Calling  
Post Handle method is Calling  
Request and Response is completed
```

# 14. Spring Boot – Servlet Filter

A filter is an object used to intercept the HTTP requests and responses of your application. By using filter, we can perform two operations at two instances:

- Before sending the request to the controller
- Before sending a response to the client.

The following code shows the sample code for a Servlet Filter implementation class with @Component annotation.

```
@Component
public class SimpleFilter implements Filter {

    @Override
    public void destroy() {}

    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain filterchain) throws IOException, ServletException {}

    @Override
    public void init(FilterConfig filterconfig) throws ServletException {}
}
```

The following example shows the code for reading the remote host and remote address from the ServletRequest object before sending the request to the controller.

In doFilter() method, we have added the System.out.println statements to print the remote host and remote address.

```
package com.tutorialspoint.demo;

import java.io.IOException;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
```



```

import org.springframework.stereotype.Component;

@Component
public class SimpleFilter implements Filter {

    @Override
    public void destroy() {}

    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain filterchain) throws IOException, ServletException {
        System.out.println("Remote Host:"+request.getRemoteHost());
        System.out.println("Remote Address:"+request.getRemoteAddr());
        filterchain.doFilter(request, response);
    }

    @Override
    public void init(FilterConfig filterconfig) throws ServletException {}
}

```

In the Spring Boot main application class file, we have added the simple REST endpoint that returns the "Hello World" string.

```

package com.tutorialspoint.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
@RestController
public class DemoApplication {

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

```

```

    @RequestMapping(value="/")
    public String hello() {
        return "Hello World";
    }
}

```

The code for Maven build – pom.xml is given below:

```

<?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>com.tutorialspoint</groupId>
    <artifactId>demo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>
    <name>demo</name>
    <description>Demo project for Spring Boot</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.8.RELEASE</version>
        <relativePath/>
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
    </properties>

    <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>

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

```

The code for Gradle Build – build.gradle is given below:

```

buildscript {
    ext {
        springBootVersion = '1.5.8.RELEASE'
    }
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-
plugin:${springBootVersion}")
    }
}

apply plugin: 'java'

```

```

apply plugin: 'eclipse'
apply plugin: 'org.springframework.boot'

group = 'com.tutorialspoint'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = 1.8
repositories {
    mavenCentral()
}
dependencies {
    compile('org.springframework.boot:spring-boot-starter-web')
    testCompile('org.springframework.boot:spring-boot-starter-test')
}

```

You can create an executable JAR file, and run the Spring Boot application by using the Maven or Gradle commands shown below:

For Maven, use the command as shown below:

```
mvn clean install
```

After BUILD SUCCESS, you can find the JAR file under the target directory.

For Gradle, use the command as shown below:

```
gradle clean build
```

After BUILD SUCCESSFUL, you can find the JAR file under the build/libs directory.

Now, run the JAR file by using the following command

```
java -jar <JARFILE>
```

You can see the application has started on the Tomcat port 8080.

```

2017-11-26 13:56:27.912 INFO 13204 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2017-11-26 13:56:27.922 INFO 13204 --- [main] com.tutorialspoint.demo.DemoApplication : Started DemoApplication in 5.961 seconds (JVM running for 8.933)

```

Now hit the URL <http://localhost:8080/> and see the output **Hello World**. It should look as shown below:



Then, you can see the Remote host and Remote address on the console log as shown below:

```
Remote Host:127.0.0.1  
Remote Address:127.0.0.1
```

# 15. Spring Boot – Tomcat Port Number

Spring Boot lets you to run the same application more than once on a different port number. In this chapter, you will learn about this in detail. Note that the default port number 8080.

## Custom Port

---

In the **application.properties** file, we can set custom port number for the property `server.port`

```
server.port=9090
```

In the **application.yml** file, you can find as follows:

```
server:  
  port: 9090
```

## Random Port

---

In the **application.properties** file, we can set random port number for the property `server.port`

```
server.port=0
```

In the **application.yml** file, you can find as follows:

```
server:  
  port: 0
```

**Note:** If the **server.port** number is 0 while starting the Spring Boot application, Tomcat uses the random port number.

# 16. Spring Boot – Rest Template

Rest Template is used to create applications that consume RESTful Web Services. You can use the **exchange()** method to consume the web services for all HTTP methods. The code given below shows how to create Bean for Rest Template to auto wiring the Rest Template object.

```
package com.tutorialspoint.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.web.client.RestTemplate;

@SpringBootApplication
public class DemoApplication {

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

    @Bean
    public RestTemplate getRestTemplate() {
        return new RestTemplate();
    }

}
```

## GET

### Consuming the GET API by using RestTemplate - exchange() method

Assume this URL <http://localhost:8080/products> returns the following JSON and we are going to consume this API response by using Rest Template using the following code:

```
[
  {
    "id": "1",
    "name": "Honey"
  },
  {
    "id": "2",
    "name": "Almond"
  }
]
```

You will have to follow the given points to consume the API:

- Autowired the Rest Template Object.
- Use HttpHeaders to set the Request Headers.
- Use HttpEntity to wrap the request object.
- Provide the URL, HttpMethod, and Return type for Exchange() method.

```
@RestController
public class ConsumeWebService {

    @Autowired
    RestTemplate restTemplate;

    @RequestMapping(value = "/template/products")
    public String getProductList() {
        HttpHeaders headers = new HttpHeaders();
        headers.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
        HttpEntity<String> entity = new HttpEntity<String>(headers);
        return restTemplate.exchange("http://localhost:8080/products",
            HttpMethod.GET, entity, String.class).getBody();
    }
}
```



## POST

### Consuming POST API by using RestTemplate - exchange() method

Assume this URL <http://localhost:8080/products> returns the response shown below, we are going to consume this API response by using the Rest Template.

The code given below is the Request body:

```
{
    "id": "3",
    "name": "Ginger"
}
```

The code given below is the Response body:

```
Product is created successfully
```

You will have to follow the points given below to consume the API:

- Autowired the Rest Template Object.
- Use the HttpHeaders to set the Request Headers.
- Use the HttpEntity to wrap the request object. Here, we wrap the Product object to send it to the request body.
- Provide the URL, HttpMethod, and Return type for exchange() method.

```
@RestController
public class ConsumeWebService {

    @Autowired
    RestTemplate restTemplate;

    @RequestMapping(value="/template/products", method=RequestMethod.POST)
    public String createProducts(@RequestBody Product product) {
        HttpHeaders headers = new HttpHeaders();
        headers.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
        HttpEntity<Product> entity = new
        HttpEntity<Product>(product,headers);
        return restTemplate.exchange("http://localhost:8080/products",
        HttpMethod.POST, entity, String.class).getBody();
    }
}
```

## PUT

### Consuming PUT API by using RestTemplate - exchange() method

Assume this URL <http://localhost:8080/products/3> returns the below response and we are going to consume this API response by using Rest Template.

The code given below is Request body:

```
{
    "name":"Indian Ginger"
}
```

The code given below is the Response body:

```
Product is updated successssfully
```

You will have to follow the points given below to consume the API:

- Autowired the Rest Template Object.
- Use HttpHeaders to set the Request Headers.
- Use HttpEntity to wrap the request object. Here, we wrap the Product object to send it to the request body.
- Provide the URL, HttpMethod, and Return type for exchange() method.

```
@RestController
public class ConsumeWebService {

    @Autowired
    RestTemplate restTemplate;

    @RequestMapping(value="/template/products/{id}", method=RequestMethod.PUT)
    public String updateProduct(@PathVariable("id") String id, @RequestBody
    Product product) {
        HttpHeaders headers = new HttpHeaders();
        headers.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
        HttpEntity<Product> entity = new
    HttpEntity<Product>(product,headers);
        return restTemplate.exchange("http://localhost:8080/products/"+id,
    HttpMethod.PUT, entity, String.class).getBody();
    }
}
```

## DELETE

### Consuming DELETE API by using RestTemplate - exchange() method

Assume this URL <http://localhost:8080/products/3> returns the response given below and we are going to consume this API response by using Rest Template.

This line of code shown below is the Response body:

Product is deleted successssfully

You will have to follow the points shown below to consume the API:

- Autowired the Rest Template Object.
  - Use HttpHeaders to set the Request Headers.
  - Use HttpEntity to wrap the request object.
- Provide the URL, HttpMethod, and Return type for exchange() method.

```
@RestController
public class ConsumeWebService {

    @Autowired
    RestTemplate restTemplate;

    @RequestMapping(value="/template/products/{id}",
method=RequestMethod.DELETE)
    public String deleteProduct(@PathVariable("id") String id) {
        HttpHeaders headers = new HttpHeaders();
        headers.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
        HttpEntity<Product> entity = new HttpEntity<Product>(headers);
        return restTemplate.exchange("http://localhost:8080/products/"+id,
HttpMethod.DELETE, entity, String.class).getBody();
    }
}
```

The complete Rest Template Controller class file is given below:

```
package com.tutorialspoint.demo.controller;

import java.util.Arrays;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpEntity;
import org.springframework.http.HttpHeaders;
```

```

import org.springframework.http.HttpMethod;
import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

import com.tutorialspoint.demo.model.Product;

@RestController
public class ConsumeWebService {

    @Autowired
    RestTemplate restTemplate;

    @RequestMapping(value = "/template/products")
    public String getProductList() {
        HttpHeaders headers = new HttpHeaders();
        headers.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
        HttpEntity<String> entity = new HttpEntity<String>(headers);
        return restTemplate.exchange("http://localhost:8080/products",
        HttpMethod.GET, entity, String.class).getBody();
    }

    @RequestMapping(value="/template/products", method=RequestMethod.POST)
    public String createProducts(@RequestBody Product product) {
        HttpHeaders headers = new HttpHeaders();
        headers.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
        HttpEntity<Product> entity = new
HttpEntity<Product>(product,headers);

        return restTemplate.exchange("http://localhost:8080/products",
        HttpMethod.POST, entity, String.class).getBody();
    }

    @RequestMapping(value="/template/products/{id}", method=RequestMethod.PUT)

```

```

        public String updateProduct(@PathVariable("id") String id, @RequestBody
        Product product) {
            HttpHeaders headers = new HttpHeaders();
            headers.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
            HttpEntity<Product> entity = new
            HttpEntity<Product>(product,headers);
            return restTemplate.exchange("http://localhost:8080/products/"+id,
            HttpMethod.PUT, entity, String.class).getBody();
        }

        @RequestMapping(value="/template/products/{id}",
        method=RequestMethod.DELETE)
        public String deleteProduct(@PathVariable("id") String id) {
            HttpHeaders headers = new HttpHeaders();
            headers.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
            HttpEntity<Product> entity = new HttpEntity<Product>(headers);
            return restTemplate.exchange("http://localhost:8080/products/"+id,
            HttpMethod.DELETE, entity, String.class).getBody();
        }
    }
}

```

The code for Spring Boot Application Class – DemoApplication.java is given below:

```

package com.tutorialspoint.demo;

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

@SpringBootApplication
public class DemoApplication {

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

}

```

The code for Maven build – pom.xml is given below:

```

<?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>com.tutorialspoint</groupId>
    <artifactId>demo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>
    <name>demo</name>
    <description>Demo project for Spring Boot</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.8.RELEASE</version>
        <relativePath/>
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
    </properties>

    <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>

```

```

    </dependencies>

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

</project>

```

The code for Gradle Build – build.gradle is given below:

```

buildscript {
    ext {
        springBootVersion = '1.5.8.RELEASE'
    }
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-
plugin:${springBootVersion}")
    }
}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'org.springframework.boot'

group = 'com.tutorialspoint'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = 1.8

```

```

repositories {
    mavenCentral()
}
dependencies {
    compile('org.springframework.boot:spring-boot-starter-web')
    testCompile('org.springframework.boot:spring-boot-starter-test')
}

```

You can create an executable JAR file, and run the Spring Boot application by using the following Maven or Gradle commands:

For Maven, you can use the command given below:

```
mvn clean install
```

After "BUILD SUCCESS", you can find the JAR file under the target directory.

For Gradle, you can use the command shown below:

```
gradle clean build
```

After "BUILD SUCCESSFUL", you can find the JAR file under build/libs directory.

Now, run the JAR file by using the following command:

```
java -jar <JARFILE>
```

Now, the application has started on the Tomcat port 8080.

```

2017-11-26 13:56:27.912 INFO 13204 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2017-11-26 13:56:27.922 INFO 13204 --- [main] com.tutorialspoint.demo.DemoApplication : Started DemoApplication in 5.961 seconds (JVM running for 6.933)

```

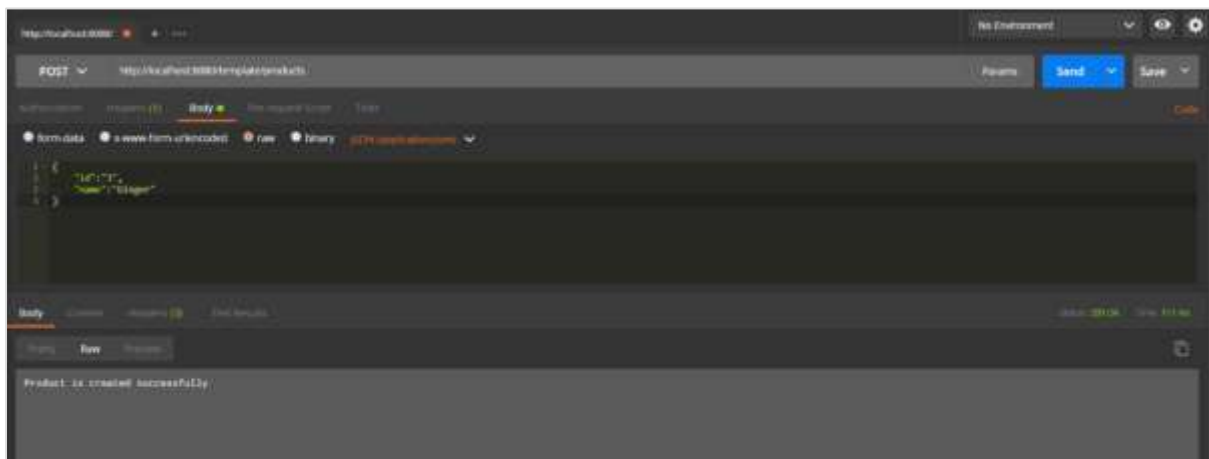
Now hit the below URL's in POSTMAN application and you can see the output.

GET Products by Rest Template - <http://localhost:8080/template/products>

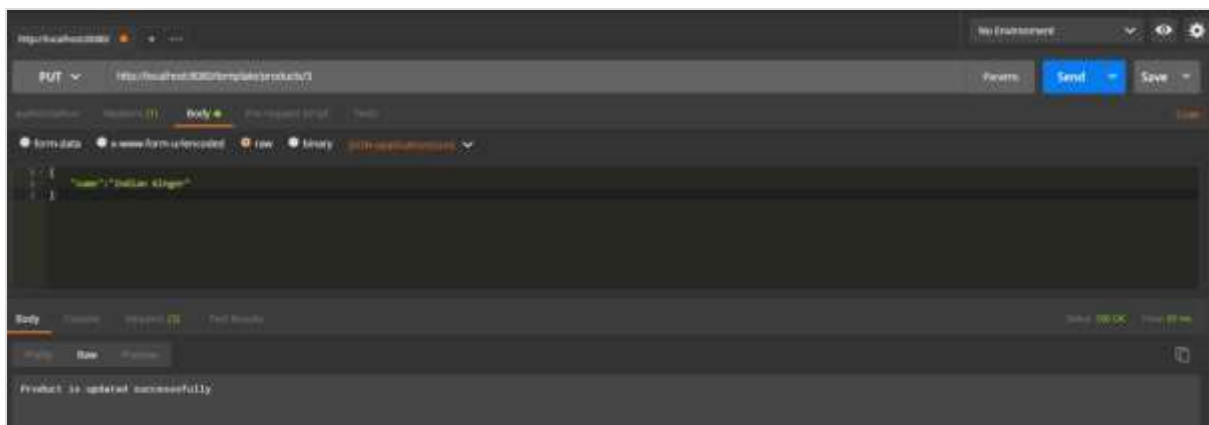




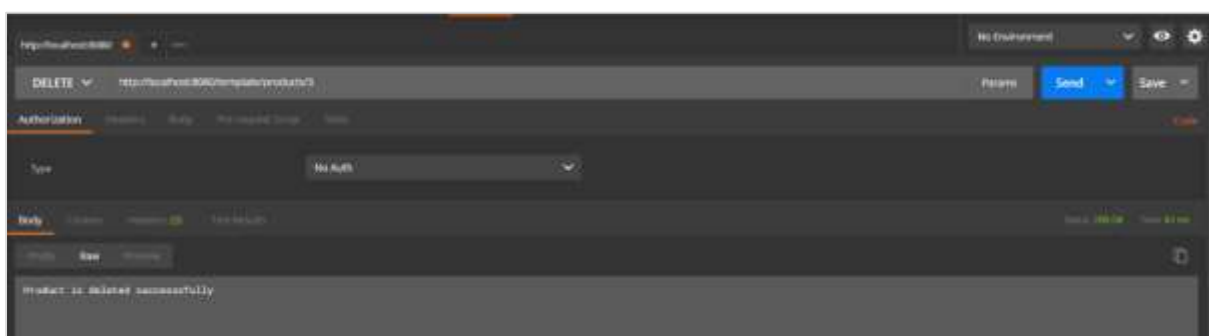
Create Products POST - <http://localhost:8080/template/products>



Update Product PUT - <http://localhost:8080/template/products/3>



Delete Product - <http://localhost:8080/template/products/3>



# 17. Spring Boot – File Handling

In this chapter, you will learn how to upload and download the file by using web service.

## File Upload

For uploading a file, you can use **MultipartFile** as a Request Parameter and this API should consume Multi-Part form data value. Observe the code given below:

```
@RequestMapping(value="/upload", method=RequestMethod.POST, consumes =
MediaType.MULTIPART_FORM_DATA_VALUE)

public String fileUpload(@RequestParam("file") MultipartFile file) {

    return null;

}
```

The complete code for the same is given below:

```
package com.tutorialspoint.demo.controller;

import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;

import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.multipart.MultipartFile;

@RestController
public class FileUploadController {

    @RequestMapping(value="/upload", method=RequestMethod.POST, consumes =
    MediaType.MULTIPART_FORM_DATA_VALUE)

    public String fileUpload(@RequestParam("file") MultipartFile file) throws
    IOException {
```

```

        File convertFile = new
File("/var/tmp/"+file.getOriginalFilename());

        convertFile.createNewFile();
        FileOutputStream fout = new FileOutputStream(convertFile);
        fout.write(file.getBytes());
        fout.close();
        return "File is upload successfully";
    }
}

```

## File Download

For file download, you should use `InputStreamResource` for downloading a File. We need to set the `HttpHeader` **Content-Disposition** in Response and need to specify the response Media Type of the application.

**Note:** In the following example, file should be available on the specified path where the application is running.

```

@RequestMapping(value="/download", method=RequestMethod.GET)
public ResponseEntity<Object> downloadFile() throws IOException {
    String filename = "/var/tmp/mysql.png";
    File file = new File(filename);

    InputStreamResource resource = new InputStreamResource(new
FileInputStream(file));

    HttpHeaders headers = new HttpHeaders();
    headers.add("Content-Disposition", String.format("attachment;
filename=\"%s\"", file.getName()));
    headers.add("Cache-Control", "no-cache, no-store, must-
revalidate");
    headers.add("Pragma", "no-cache");
    headers.add("Expires", "0");

    ResponseEntity<Object> responseEntity =
ResponseEntity.ok().headers(headers).contentType(M
ediaType.parseMediaType("application/txt")).body(resource);
    return responseEntity;
}

```

The complete code for the same is given below:

```
package com.tutorialspoint.demo.controller;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import org.springframework.core.io.InputStreamResource;
import org.springframework.http.HttpHeaders;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class FileDownloadController {

    @RequestMapping(value="/download", method=RequestMethod.GET)
    public ResponseEntity<Object> downloadFile() throws IOException {
        String filename = "/var/tmp/mysql.png";
        File file = new File(filename);
        InputStreamResource resource = new InputStreamResource(new
        FileInputStream(file));
        HttpHeaders headers = new HttpHeaders();
        headers.add("Content-Disposition", String.format("attachment;
        filename=\"%s\"", file.getName()));
        headers.add("Cache-Control", "no-cache, no-store, must-
        revalidate");
        headers.add("Pragma", "no-cache");
        headers.add("Expires", "0");

        ResponseEntity<Object> responseEntity =
        ResponseEntity.ok().headers(headers).contentType(M
        ediaType.parseMediaType("application/txt")).body(resource);

        return responseEntity;
    }
}
```

The main Spring Boot application is given below

```
package com.tutorialspoint.demo;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

The code for Maven build – pom.xml is given below:

```
<?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>com.tutorialspoint</groupId>
    <artifactId>demo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>
    <name>demo</name>
    <description>Demo project for Spring Boot</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.8.RELEASE</version>
        <relativePath/>
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
```

```

</properties>

<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>

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

</project>

```

The code for Gradle Build – build.gradle is given below:

```

buildscript {
    ext {
        springBootVersion = '1.5.8.RELEASE'
    }
    repositories {
        mavenCentral()
    }
    dependencies {

```

```

        classpath("org.springframework.boot:spring-boot-gradle-
plugin:${springBootVersion}")
    }
}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'org.springframework.boot'

group = 'com.tutorialspoint'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = 1.8

repositories {
    mavenCentral()
}

dependencies {
    compile('org.springframework.boot:spring-boot-starter-web')
    testCompile('org.springframework.boot:spring-boot-starter-test')
}

```

Now you can create an executable JAR file, and run the Spring Boot application by using the Maven or Gradle commands given below:

For Maven, use the command given below:

```
mvn clean install
```

After "BUILD SUCCESS", you can find the JAR file under target directory.

For Gradle, you can use the command shown below:

```
sgradle clean build
```

After "BUILD SUCCESSFUL", you can find the JAR file under build/libs directory.

Now, run the JAR file by using the following command:

```
java -jar <JARFILE>
```

This will start the application on the Tomcat port 8080 as shown below:

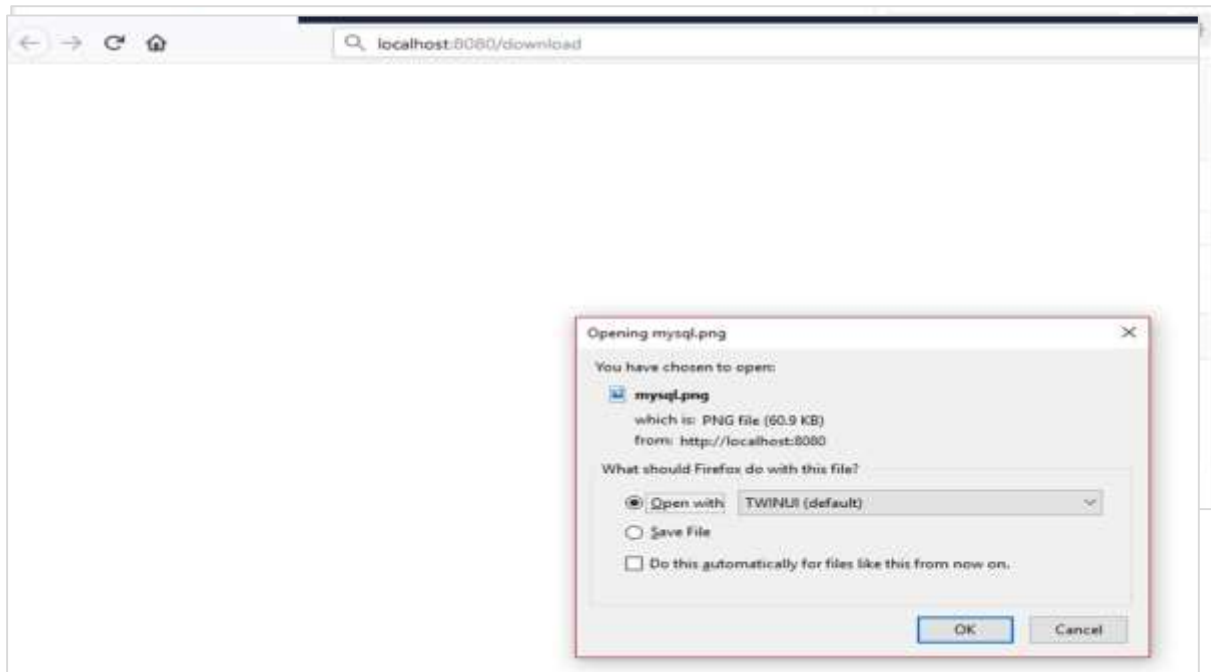
```

2017-11-26 13:56:27.912 INFO 13204 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2017-11-26 13:56:27.922 INFO 13204 --- [main] com.tutorialspoint.demo.DemoApplication : Started DemoApplication in 5.961 seconds (JVM running for 6.933)

```

Now hit the below URL's in POSTMAN application and you can see the output as shown below:

File upload - <http://localhost:8080/upload>



File download - <http://localhost:8080/upload>



# 18. Spring Boot – Service Components

Service Components are the class file which contains @Service annotation. These class files are used to write business logic in a different layer, separated from @RestController class file. The logic for creating a service component class file is shown here:

```
public interface ProductService {  
}
```

The class that implements the Interface with @Service annotation is as shown:

```
@Service  
public class ProductServiceImpl implements ProductService {  
}
```

Observe that in this tutorial, we are using **Product Service API(s)** to store, retrieve, update and delete the products. We wrote the business logic in @RestController class file itself. Now, we are going to move the business logic code from controller to service component.

You can create an Interface which contains add, edit, get and delete methods using the code as shown below:

```
package com.tutorialspoint.demo.service;  
  
import java.util.Collection;  
  
import com.tutorialspoint.demo.model.Product;  
  
public interface ProductService {  
  
    public abstract void createProduct(Product product);  
  
    public abstract void updateProduct(String id, Product product);  
  
    public abstract void deleteProduct(String id);  
  
    public abstract Collection<Product> getProducts();  
  
}
```

The following code will let you to create a class which implements the ProductService interface with @Service annotation and write the business logic to store, retrieve, delete and updates the product.

```
package com.tutorialspoint.demo.service;

import java.util.Collection;
import java.util.HashMap;
import java.util.Map;
import org.springframework.stereotype.Service;
import com.tutorialspoint.demo.model.Product;

@Service
public class ProductServiceImpl implements ProductService {

    private static Map<String, Product> productRepo = new HashMap<>();

    static {

        Product honey = new Product();
        honey.setId("1");
        honey.setName("Honey");
        productRepo.put(honey.getId(), honey);

        Product almond = new Product();
        almond.setId("2");
        almond.setName("Almond");
        productRepo.put(almond.getId(), almond);

    }

    @Override
    public void createProduct(Product product) {
        productRepo.put(product.getId(), product);
    }

    @Override
    public void updateProduct(String id, Product product) {
        productRepo.remove(id);
        product.setId(id);
        productRepo.put(id, product);
    }
}
```

```

    }

    @Override
    public void deleteProduct(String id) {
        productRepo.remove(id);
    }

    @Override
    public Collection<Product> getProducts() {
        return productRepo.values();
    }
}

```

The code here show the Rest Controller class file, here we @Autowired the ProductService interface and called the methods.

```

package com.tutorialspoint.demo.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

import com.tutorialspoint.demo.model.Product;
import com.tutorialspoint.demo.service.ProductService;

@RestController
public class ProductServiceController {

    @Autowired
    ProductService productService;
}

```

```

    @RequestMapping(value = "/products")
    public ResponseEntity<Object> getProduct() {
        return new ResponseEntity<>(productService.getProducts(),
        HttpStatus.OK);
    }

    @RequestMapping(value = "/products/{id}", method = RequestMethod.PUT)
    public ResponseEntity<Object> updateProduct(@PathVariable("id") String id,
    @RequestBody Product product) {
        productService.updateProduct(id, product);
        return new ResponseEntity<>("Product is updated successssfully",
        HttpStatus.OK);
    }

    @RequestMapping(value = "/products/{id}", method = RequestMethod.DELETE)
    public ResponseEntity<Object> delete(@PathVariable("id") String id) {
        productService.deleteProduct(id);
        return new ResponseEntity<>("Product is deleted successssfully",
        HttpStatus.OK);
    }

    @RequestMapping(value = "/products", method = RequestMethod.POST)
    public ResponseEntity<Object> createProduct(@RequestBody Product product)
    {
        productService.createProduct(product);
        return new ResponseEntity<>("Product is created successfully",
        HttpStatus.CREATED);
    }
}

```

The code for POJO class – Product.java is shown here:

```

package com.tutorialspoint.demo.model;

public class Product {
    private String id;
    private String name;

    public String getId() {

```

```

        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

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

```

A main Spring Boot application is given below

```

package com.tutorialspoint.demo;

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

@SpringBootApplication
public class DemoApplication {

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

}

```

The code for Maven build – pom.xml is shown below:

```

<?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>com.tutorialspoint</groupId>
<artifactId>demo</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>
<name>demo</name>
<description>Demo project for Spring Boot</description>

<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.5.8.RELEASE</version>
    <relativePath/>
</parent>

<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
    <java.version>1.8</java.version>
</properties>

<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>

<build>
    <plugins>

```

```

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

</project>

```

The code for Gradle Build – build.gradle is shown below:

```

buildscript {
    ext {
        springBootVersion = '1.5.8.RELEASE'
    }
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-
plugin:${springBootVersion}")
    }
}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'org.springframework.boot'

group = 'com.tutorialspoint'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = 1.8

repositories {
    mavenCentral()
}

dependencies {
    compile('org.springframework.boot:spring-boot-starter-web')
}

```

```
testCompile('org.springframework.boot:spring-boot-starter-test')
}
```

You can create an executable JAR file, and run the Spring Boot application by using the Maven or Gradle commands given below:

For Maven, use the command as shown below:

```
mvn clean install
```

After "BUILD SUCCESS", you can find the JAR file under the target directory.

For Gradle, you can use the command as shown below:

```
gradle clean build
```

After "BUILD SUCCESSFUL", you can find the JAR file under build/libs directory.

Run the JAR file by using the command given below:

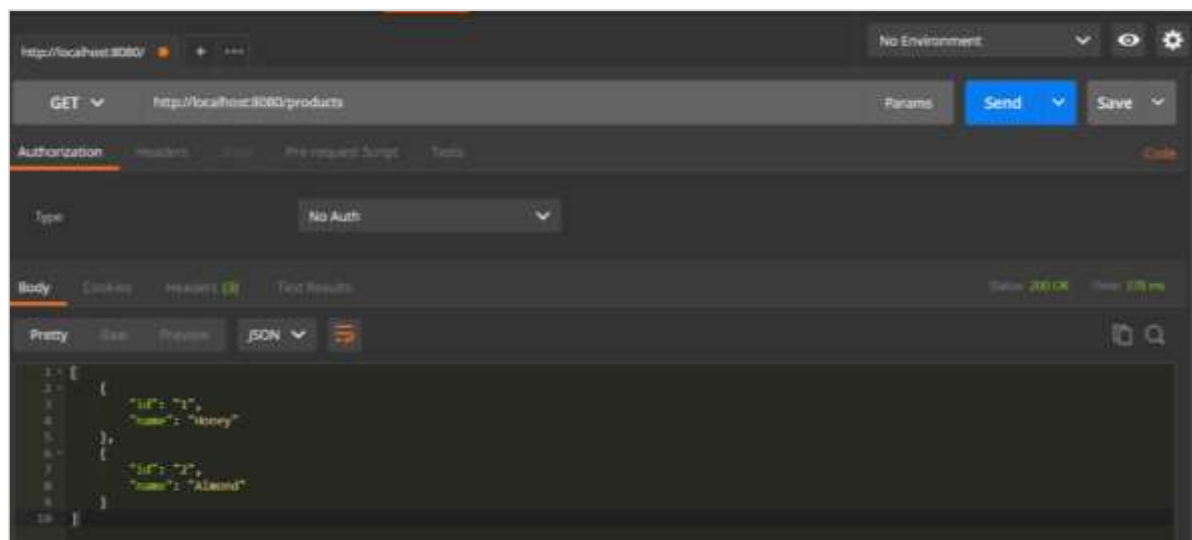
```
java -jar <JARFILE>
```

Now, the application has started on the Tomcat port 8080 as shown in the image given below:

```
2017-11-26 13:56:27.912 INFO 13204 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2017-11-26 13:56:27.922 INFO 13204 --- [main] com.tutorialspoint.demo.DemoApplication : Started DemoApplication in 5.961 seconds (JVM running for 6.933)
```

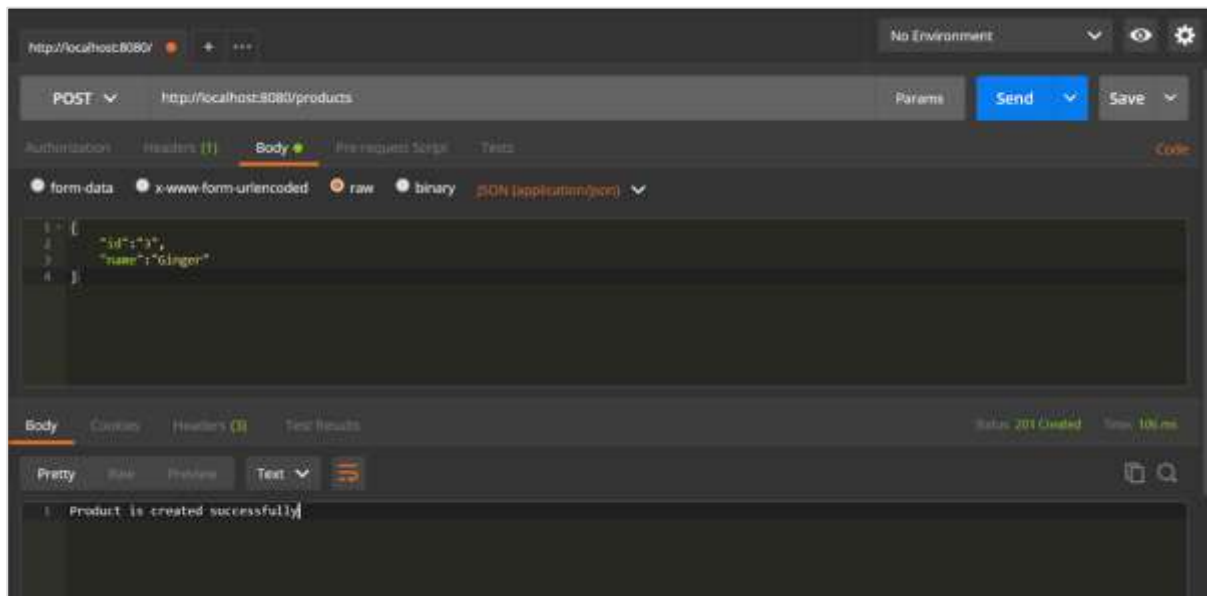
Now hit the below URL's in POSTMAN application and you can see the output as shown below:

GET API URL is: <http://localhost:8080/products>

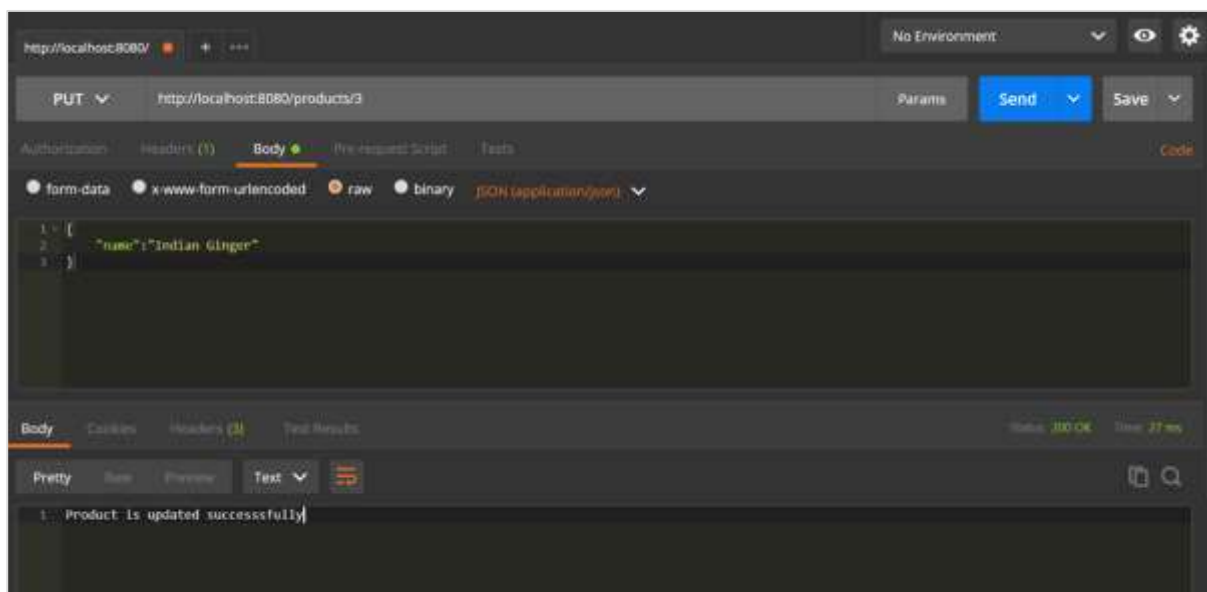




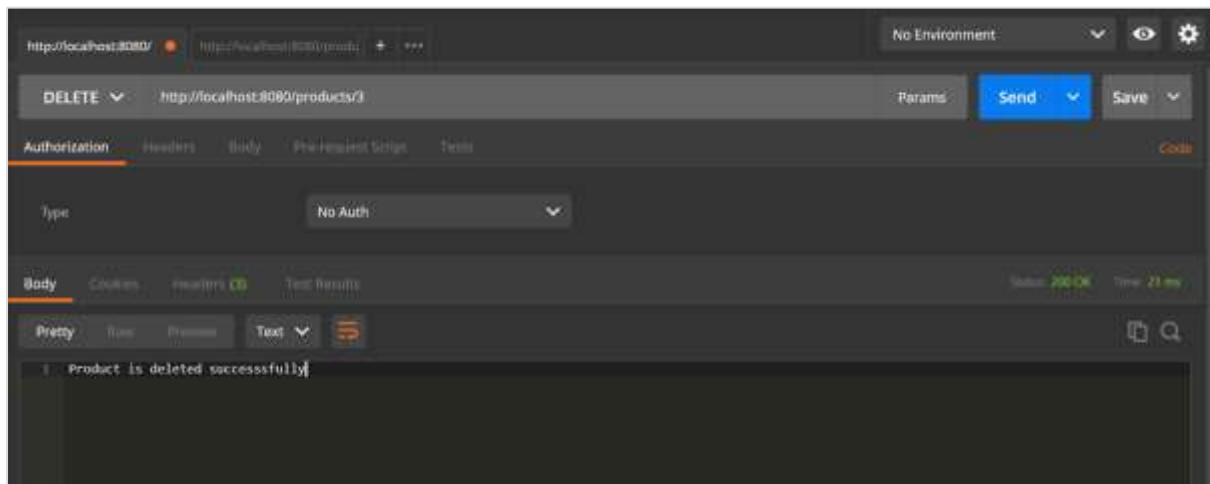
POST API URL is: <http://localhost:8080/products>



PUT API URL is: <http://localhost:8080/products/3>



DELETE API URL is: <http://localhost:8080/products/3>



# 19. Spring Boot – Thymeleaf

Thymeleaf is a Java-based library used to create a web application. It provides a good support for serving a XHTML/HTML5 in web applications. In this chapter, you will learn in detail about Thymeleaf.

## Thymeleaf Templates

---

Thymeleaf converts your files into well-formed XML files. It contains 6 types of templates as given below:

- XML
- Valid XML
- XHTML
- Valid XHTML
- HTML5
- Legacy HTML5

All templates, except Legacy HTML5, are referring to well-formed valid XML files. Legacy HTML5 allows us to render the HTML5 tags in web page including not closed tags.

## Web Application

---

You can use Thymeleaf templates to create a web application in Spring Boot. You will have to follow the below steps to create a web application in Spring Boot by using Thymeleaf.

Use the following code to create a @Controller class file to redirect the Request URI to HTML file:

```
package com.tutorialspoint.demo.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class WebController {

    @RequestMapping(value = "/index")
    public String index() {
        return "index";
    }

}
```

In the above example, the request URI is **/index**, and the control is redirected into the index.html file. Note that the index.html file should be placed under the templates directory and all JS and CSS files should be placed under the static directory in classpath. In the example shown, we used CSS file to change the color of the text.

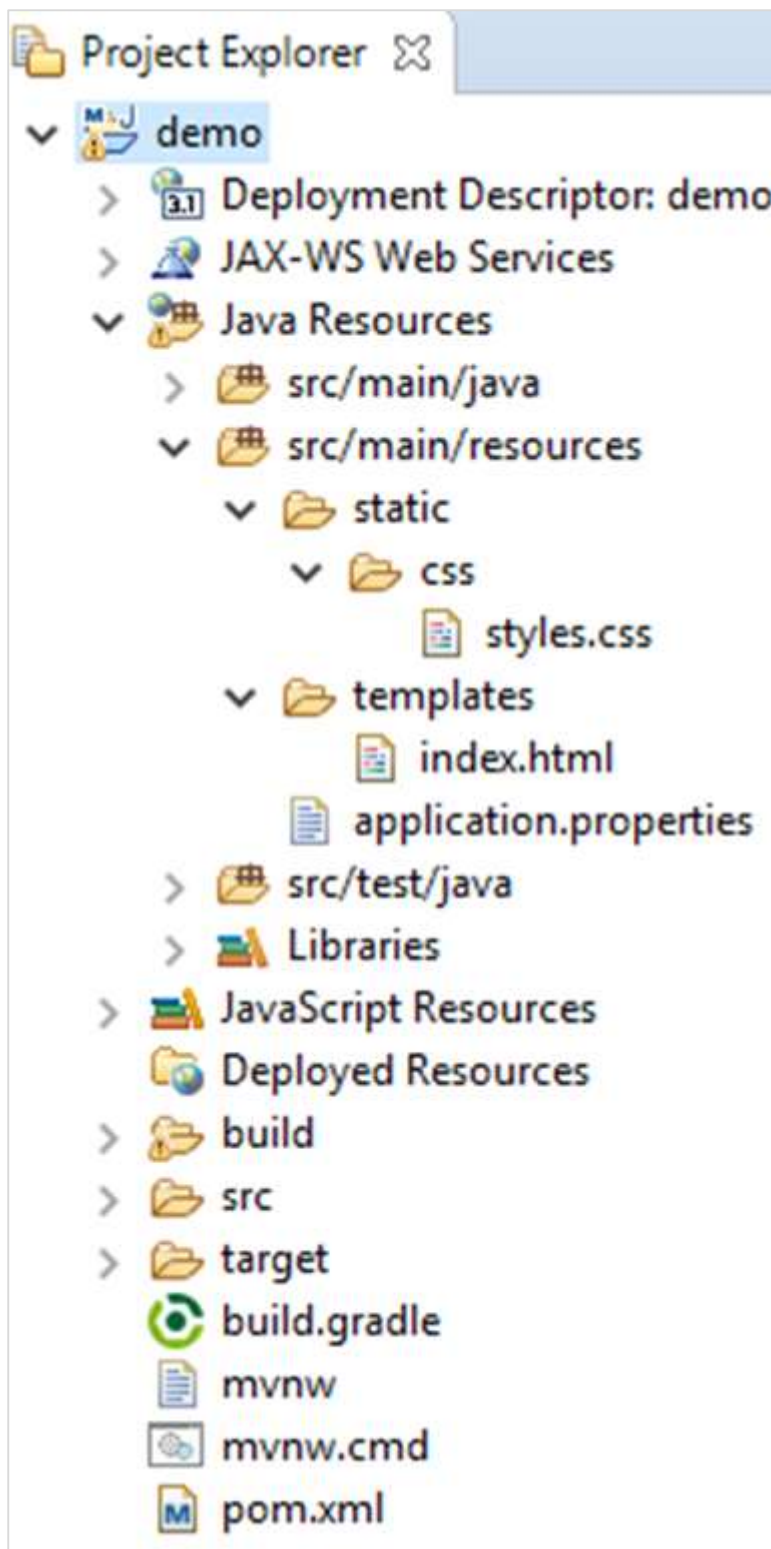
You can use the following code and created a CSS file in separate folder **css** and name the file as styles.css:

```
h4 {  
    color: red;  
}
```

The code for index.html file is given below:

```
<!DOCTYPE html>  
<html>  
  <head>  
    <meta charset="ISO-8859-1" />  
    <link href="css/styles.css" rel="stylesheet"/>  
    <title>Spring Boot Application</title>  
  </head>  
  <body>  
    <h4>Welcome to Thymeleaf Spring Boot web application</h4>  
  </body>  
</html>
```

The project explorer is shown in the screenshot given below:



Now, we need to add the Spring Boot Starter Thymeleaf dependency in our build configuration file.

Maven users can add the following dependency into the pom.xml file:

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

Gradle users can add the following dependency in the build.gradle file:

```
compile group: 'org.springframework.boot', name: 'spring-boot-starter-
thymeleaf'
```

The code for main Spring Boot application class file is given below:

```
package com.tutorialspoint.demo;

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

@SpringBootApplication
public class DemoApplication {

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

}
```

The code for Maven – pom.xml is given below:

```
<?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>com.tutorialspoint</groupId>
  <artifactId>demo</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>demo</name>
```

```

<description>Demo project for Spring Boot</description>

<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.5.8.RELEASE</version>
    <relativePath />
</parent>

<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
    <java.version>1.8</java.version>
</properties>

<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>

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

</dependencies>

<build>
    <plugins>

```

```

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

</project>

```

The code for Gradle – build.gradle is given below:

```

buildscript {
    ext {
        springBootVersion = '1.5.8.RELEASE'
    }
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-
plugin:${springBootVersion}")
    }
}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'org.springframework.boot'

group = 'com.tutorialspoint'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = 1.8

repositories {
    mavenCentral()
}

```



```
dependencies {
    compile('org.springframework.boot:spring-boot-starter-web')
    compile group: 'org.springframework.boot', name: 'spring-boot-starter-thymeleaf'
    testCompile('org.springframework.boot:spring-boot-starter-test')
}
```

You can create an executable JAR file, and run the spring boot application by using the following Maven or Gradle commands:

For Maven, use the command as shown below:

```
mvn clean install
```

After "BUILD SUCCESS", you can find the JAR file under the target directory.

For Gradle, use the command as shown below:

```
gradle clean build
```

After "BUILD SUCCESSFUL", you can find the JAR file under the build/libs directory.

Run the JAR file by using the command given here:

```
java -jar <JARFILE>
```

Now, the application has started on the Tomcat port 8080 as shown below:

```
2017-11-26 13:56:27.912 INFO 13204 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2017-11-26 13:56:27.922 INFO 13204 --- [main] com.tutorialspoint.demo.DemoApplication : Started DemoApplication in 5.961 seconds (JVM running for 6.933)
```

Now hit the URL in your web browser and you can see the output as shown:

<http://localhost:8080/index>



## 20. Spring Boot – Consuming RESTful Web Services

This chapter will discuss in detail about consuming a RESTful Web Services by using jQuery AJAX.

Create a simple Spring Boot web application and write a controller class files which is used to redirects into the HTML file to consumes the RESTful web services.

We need to add the Spring Boot starter Thymeleaf and Web dependency in our build configuration file.

For Maven users, add the below dependencies in your pom.xml file.

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

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

For Gradle users, add the below dependencies into your build.gradle file:

```
compile group: 'org.springframework.boot', name: 'spring-boot-starter-
thymeleaf'
compile('org.springframework.boot:spring-boot-starter-web')
```

The code for @Controller class file is given below:

```
@Controller
public class ViewController {
}
```

You can define the Request URI methods to redirects into the HTML file as shown below:

```
@RequestMapping("/view-products")
public String viewProducts() {
    return "view-products";}

@RequestMapping("/add-products")
public String addProducts() {
    return "add-products";}
```

This API <http://localhost:9090/products> should return the below JSON in response as shown below:

```
[
  {
    "id": "1",
    "name": "Honey"
  },
  {
    "id": "2",
    "name": "Almond"
  }
]
```

Now, create a view-products.html file under the templates directory in the classpath.

In the HTML file, we added the jQuery library and written the code to consume the RESTful web service on page load.

```
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.js"></script>
<script>
$(document).ready(function(){
    $.getJSON("http://localhost:9090/products", function(result){
        $.each(result, function(key,value) {
            $("#productsJson").append(value.id+" "+value.name+" ");
        });
    });
});
</script>
```

The POST method and this URL <http://localhost:9090/products> should contains the below Request Body and Response body.

The code for Request body is given below:

```
{
  "id": "3",
  "name": "Ginger"
}
```

The code for Response body is given below:

Product is created successfully

Now, create the add-products.html file under the templates directory in the classpath.

In the HTML file, we added the jQuery library and written the code that submits the form to RESTful web service on clicking the button

```
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.js"></script
>
<script>
    $(document).ready(function() {
        $("button").click(function() {
            var productmodel = {
                id : "3",
                name : "Ginger"
            };
            var requestJSON = JSON.stringify(productmodel);
            $.ajax({
                type : "POST",
                url : "http://localhost:9090/products",
                headers : {
                    "Content-Type" : "application/json"
                },
                data : requestJSON,
                success : function(data) {
                    alert(data);
                },
                error : function(data) {
                }
            });
        });
    });
</script>
```

The complete code is given below.

Maven – pom.xml file

```
<?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>com.tutorialspoint</groupId>
    <artifactId>demo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>
    <name>demo</name>
    <description>Demo project for Spring Boot</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.8.RELEASE</version>
        <relativePath />
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
    </properties>

    <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>

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

</dependencies>

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

</project>

```

The code for Gradle – build.gradle is given below:

```

buildscript {
    ext {
        springBootVersion = '1.5.8.RELEASE'
    }
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-
plugin:${springBootVersion}")
    }
}

```

```

}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'org.springframework.boot'

group = 'com.tutorialspoint'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = 1.8

repositories {
    mavenCentral()
}

dependencies {
    compile('org.springframework.boot:spring-boot-starter-web')
    compile group: 'org.springframework.boot', name: 'spring-boot-starter-thymeleaf'
    testCompile('org.springframework.boot:spring-boot-starter-test')
}

```

The controller class file given below – ViewController.java is given below:

```

package com.tutorialspoint.demo.controller;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
public class ViewController {
    @RequestMapping("/view-products")
    public String viewProducts() {
        return "view-products";
    }

    @RequestMapping("/add-products")
    public String addProducts() {
        return "add-products";    }
}

```

The view-products.html file is given below:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="ISO-8859-1"/>
    <title>View Products</title>
  </head>
  <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.js"></script>
  </script>
  <script>
$(document).ready(function(){
  $.getJSON("http://localhost:9090/products", function(result){
    $.each(result, function(key,value) {
      $("#productsJson").append(value.id+" "+value.name+" ");
    });
  });
});
</script>
  </head>
  <body>
    <div id="productsJson">
    </div>
  </body>
</html>
```

The add-products.html file is given below:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1" />
<title>Add Products</title>
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.js"></s
cript>
<script>
$(document).ready(function() {
  $("button").click(function() {
```



```

        var productmodel = {
            id : "3",
            name : "Ginger"
        };
        var requestJSON = JSON.stringify(productmodel);
        $.ajax({
            type : "POST",
            url : "http://localhost:9090/products",
            headers : {
                "Content-Type" : "application/json"
            },
            data : requestJSON,
            success : function(data) {
                alert(data);
            },
            error : function(data) {
            }
        });
    });
});
</script>
</head>
<body>
    <button>Click here to submit the form</button></body> </html>

```

The main Spring Boot Application class file is given below:

```

package com.tutorialspoint.demo;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

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

```

Now, you can create an executable JAR file, and run the Spring Boot application by using the following Maven or Gradle commands.

For Maven, use the command as given below:

```
mvn clean install
```

After "BUILD SUCCESS", you can find the JAR file under the target directory.

For Gradle, use the command as given below:

```
gradle clean build
```

After "BUILD SUCCESSFUL", you can find the JAR file under the build/libs directory.

Run the JAR file by using the following command:

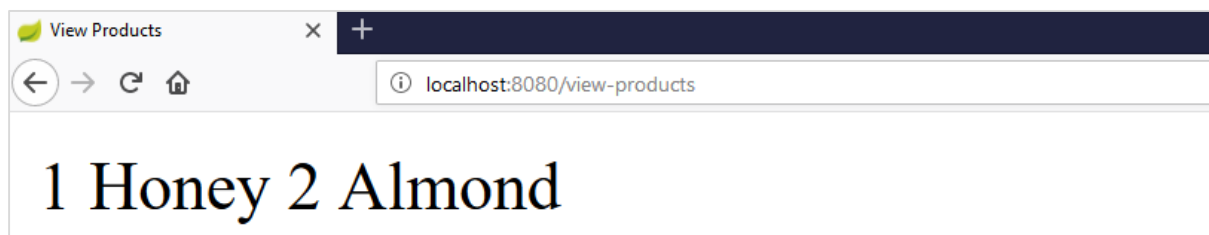
```
java -jar <JARFILE>
```

Now, the application has started on the Tomcat port 8080.

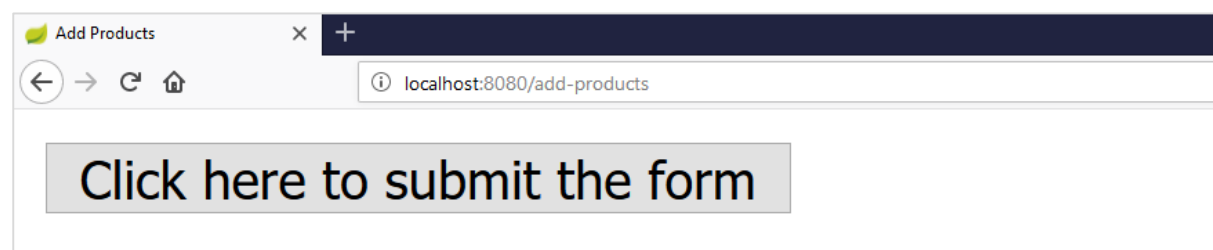
```
2017-11-26 13:56:27.912 INFO 13204 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2017-11-26 13:56:27.922 INFO 13204 --- [main] com.tutorialspoint.demo.DemoApplication : Started DemoApplication in 5.961 seconds (3V
M running for 6.933)
```

Now hit the URL in your web browser and you can see the output as shown:

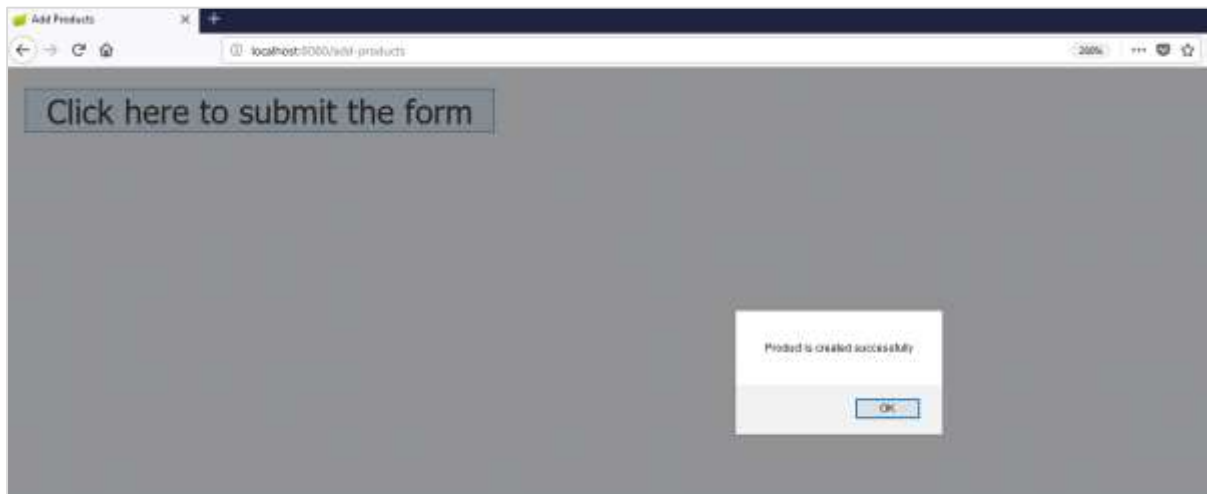
<http://localhost:8080/view-products>



<http://localhost:8080/add-products>



Now, click the button **Click here to submit the form** and you can see the result as shown:



Now, hit the view products URL and see the created product.

<http://localhost:8080/view-products>



## Angular JS

To consume the APIs by using Angular JS, you can use the examples given below:

Use the following code to create the Angular JS Controller to consume the GET API - <http://localhost:9090/products> :

```
angular.module('demo', [])
.controller('Hello', function($scope, $http) {
    $http.get('http://localhost:9090/products').
        then(function(response) {
            $scope.products = response.data;
        });
});
```

Use the following code to create the Angular JS Controller to consume the POST API - <http://localhost:9090/products> :

```
angular.module('demo', [])  
.controller('Hello', function($scope, $http) {  
    $http.post('http://localhost:9090/products',data).  
        then(function(response) {  
            console.log("Product created successfully");  
        });  
});
```

**Note:** The Post method data represents the Request body in JSON format to create a product.

# 21. Spring Boot – CORS Support

Cross-Origin Resource Sharing (CORS) is a security concept that allows restricting the resources implemented in web browsers. It prevents the JavaScript code producing or consuming the requests against different origin.

For example, your web application is running on 8080 port and by using JavaScript you are trying to consuming RESTful web services from 9090 port. Under such situations, you will face the Cross-Origin Resource Sharing security issue on your web browsers.

Two requirements are needed to handle this issue:

- RESTful web services should support the Cross-Origin Resource Sharing.
- RESTful web service application should allow accessing the API(s) from the 8080 port.

In this chapter, we are going to learn in detail about How to Enable Cross-Origin Requests for a RESTful Web Service application.

## Enable CORS in Controller Method

We need to set the origins for RESTful web service by using **@CrossOrigin** annotation for the controller method. This **@CrossOrigin** annotation supports specific REST API, and not for the entire application.

```
@RequestMapping(value = "/products")
@CrossOrigin(origins = "http://localhost:8080")
public ResponseEntity<Object> getProduct() {
    return null;
}
```

## Global CORS Configuration

We need to define the shown **@Bean** configuration to set the CORS configuration support globally to your Spring Boot application.

```
@Bean
public WebMvcConfigurer corsConfigurer() {
    return new WebMvcConfigurerAdapter() {
        @Override
        public void addCorsMappings(CorsRegistry registry) {
            registry.addMapping("/products").allowedOrigins("http://localhost:9000");
        }
    };
}
```

```
}    };
```

To code to set the CORS configuration globally in main Spring Boot application is given below.

```
package com.tutorialspoint.demo;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.web.servlet.config.annotation.CorsRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
import
org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;

@SpringBootApplication
public class DemoApplication {

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

    @Bean
    public WebMvcConfigurer corsConfigurer() {
        return new WebMvcConfigurerAdapter() {
            @Override
            public void addCorsMappings(CorsRegistry registry) {
                registry.addMapping("/products").allowedOrigins("http://localhost:8080");
            }
        };
    }
}
```

Now, you can create a Spring Boot web application that runs on 8080 port and your RESTful web service application that can run on the 9090 port. For further details about implementation about RESTful Web Service, you can refer to the chapter titled **Consuming RESTful Web Services** of this tutorial.

## 22. Spring Boot – Internationalization

Internationalization is a process that makes your application adaptable to different languages and regions without engineering changes on the source code. In other words, Internationalization is a readiness of Localization.

In this chapter, we are going to learn in detail about How to implement the Internationalization in Spring Boot.

### Dependencies

---

We need the Spring Boot Starter Web and Spring Boot Starter Thymeleaf dependency to develop a web application in Spring Boot.

#### Maven

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

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

#### Gradle

```
compile('org.springframework.boot:spring-boot-starter-web')
compile group: 'org.springframework.boot', name: 'spring-boot-starter-
thymeleaf'
```

### LocaleResolver

---

We need to determine default Locale of your application. We need to add the LocaleResolver bean in our Spring Boot application.

```
@Bean
public LocaleResolver localeResolver() {
    SessionLocaleResolver sessionLocaleResolver = new SessionLocaleResolver();
    sessionLocaleResolver.setDefaultLocale(Locale.US);
    return sessionLocaleResolver;}
```

## LocaleChangeInterceptor

LocaleChangeInterceptor is used to change the new Locale based on the value of the language parameter added to a request.

```
@Bean
public LocaleChangeInterceptor localeChangeInterceptor() {
    LocaleChangeInterceptor localeChangeInterceptor = new
    LocaleChangeInterceptor();
    localeChangeInterceptor.setParamName("language");
    return localeChangeInterceptor;
}
```

To take this effect, we need to add the LocaleChangeInterceptor into the application's registry interceptor. The configuration class should extend the WebMvcConfigurerAdapter class and override the addInterceptors() method.

```
@Override
public void addInterceptors(InterceptorRegistry registry) {
    registry.addInterceptor(localeChangeInterceptor());
}
```

## Messages Sources

Spring Boot application by default takes the message sources from **src/main/resources** folder under the classpath. The default locale message file name should be **message.properties** and files for each locale should name as **messages\_XX.properties**. The "XX" represents the locale code.

All the message properties should be used as key pair values. If any properties are not found on the locale, the application uses the default property from messages.properties file.

The default messages.properties will be as shown:

```
welcome.text=Hi Welcome to Everyone
```

The French language messages\_fr.properties will be as shown:

```
welcome.text=Salut Bienvenue à tous
```

Note: Messages source file should be saved as "UTF-8" file format.



## HTML file

In the HTML file, use the syntax **`#{key}`** to display the messages from the properties file.

```
<h1 th:text="#{welcome.text}"></h1>
```

The complete code is given below

### Maven – pom.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>com.tutorialspoint</groupId>
    <artifactId>demo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>
    <name>demo</name>
    <description>Demo project for Spring Boot</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.8.RELEASE</version>
        <relativePath />
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
    </properties>

    <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>

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

</dependencies>

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

```

### Gradle – build.gradle

```

buildscript {
    ext {
        springBootVersion = '1.5.8.RELEASE'
    }
    repositories {
        mavenCentral()
    }
    dependencies {

```

```

classpath("org.springframework.boot:spring-boot-gradle-
plugin:${springBootVersion}")

    }
}
apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'org.springframework.boot'

group = 'com.tutorialspoint'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = 1.8
repositories {
    mavenCentral()
}
dependencies {
    compile('org.springframework.boot:spring-boot-starter-web')
    compile group: 'org.springframework.boot', name: 'spring-boot-starter-
thymeleaf'
    testCompile('org.springframework.boot:spring-boot-starter-test')
}

```

The main Spring Boot application class file is given below

```

package com.tutorialspoint.demo;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}

```

The controller class file is given below:

```
package com.tutorialspoint.demo.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class ViewController {
    @RequestMapping("/locale")
    public String locale() {
        return "locale";
    }
}
```

Configuration class to support the Internationalization

```
package com.tutorialspoint.demo;

import java.util.Locale;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.LocaleResolver;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;
import org.springframework.web.servlet.i18n.LocaleChangeInterceptor;
import org.springframework.web.servlet.i18n.SessionLocaleResolver;

@Configuration
public class Internationalization extends WebMvcConfigurerAdapter {

    @Bean
    public LocaleResolver localeResolver() {
        SessionLocaleResolver sessionLocaleResolver = new
SessionLocaleResolver();
        sessionLocaleResolver.setDefaultLocale(Locale.US);
        return sessionLocaleResolver;
    }
}
```

```

@Bean
public LocaleChangeInterceptor localeChangeInterceptor() {
    LocaleChangeInterceptor localeChangeInterceptor = new
LocaleChangeInterceptor();
    localeChangeInterceptor.setParamName("language");
    return localeChangeInterceptor;
}

@Override
public void addInterceptors(InterceptorRegistry registry) {
    registry.addInterceptor(localeChangeInterceptor());
}
}

```

The Message sources – messages.properties is as shown:

```
welcome.text=Hi Welcome to Everyone
```

The Message sources – message\_fr.properties is as shown:

```
welcome.text=Salut Bienvenue à tous
```

The HTML file locale.html should be placed under the templates directory on the classpath as shown:

```

<!DOCTYPE html>
<html>
    <head>
        <meta charset="ISO-8859-1"/>
        <title>Internationalization</title>
    </head>
    <body>
        <h1 th:text="#{welcome.text}"></h1>
    </body>
</html>

```

You can create an executable JAR file, and run the Spring boot application by using the following Maven or Gradle commands:

For Maven, use the following command:

```
mvn clean install
```

After "BUILD SUCCESS", you can find the JAR file under the target directory.

For Gradle, use the following command:

```
gradle clean build
```

After "BUILD SUCCESSFUL", you can find the JAR file under the build/libs directory.

Now, run the JAR file by using the command as shown:

```
java -jar <JARFILE>
```

You will find that the application has started on the Tomcat port 8080.

```
2017-11-26 13:56:27.912 INFO 13204 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2017-11-26 13:56:27.922 INFO 13204 --- [main] com.tutorialspoint.demo.DemoApplication : Started DemoApplication in 5.961 seconds (3V
M running for 6.933)
```

Now hit the URL <http://localhost:8080/locale> in your web browser and you can see the following output:



The URL <http://localhost:8080/locale?language=fr> will give you the output as shown:



# 23. Spring Boot – Scheduling

Scheduling is a process of executing the tasks for the specific time period. Spring Boot provides a good support to write a scheduler on the Spring applications.

## Java Cron Expression

Java Cron expressions are used to configure the instances of CronTrigger, a subclass of org.quartz.Trigger. For more information about Java cron expression you can refer to this link:

[https://docs.oracle.com/cd/E12058\\_01/doc/doc.1014/e12030/cron\\_expressions.htm](https://docs.oracle.com/cd/E12058_01/doc/doc.1014/e12030/cron_expressions.htm)

The @EnableScheduling annotation is used to enable the scheduler for your application. This annotation should be added into the main Spring Boot application class file.

```
@SpringBootApplication
@EnableScheduling
public class DemoApplication {

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

The @Scheduled annotation is used to trigger the scheduler for a specific time period.

```
@Scheduled(cron = "0 * 9 * * ?")
public void cronJobSch() throws Exception {
}
```

The following is a sample code that shows how to execute the task every minute starting at 9:00 AM and ending at 9:59 AM, every day

```
package com.tutorialspoint.demo.scheduler;
import java.text.SimpleDateFormat;
import java.util.Date;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;
@Component
```

```

public class Scheduler {
    @Scheduled(cron = "0 * 9 * * ?")
    public void cronJobSch() {
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss.SSS");
        Date now = new Date();
        String strDate = sdf.format(now);
        System.out.println("Java cron job expression:: " + strDate);
    }
}

```

The following screenshot shows how the application has started at 09:03:23 and for every one minute from that time the cron job scheduler task has executed.

```

2017-12-06 09:03:23.749 INFO 16232 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2017-12-06 09:03:23.758 INFO 16232 --- [main] com.tutorialspoint.demo.DemoApplication : Started DemoApplication in 6.762 seconds (JVM
% running for 7.612)
Java cron job expression:: 2017-12-06 09:04:00.002
Java cron job expression:: 2017-12-06 09:05:00.002
Java cron job expression:: 2017-12-06 09:06:00.001

```

## Fixed Rate

Fixed Rate scheduler is used to execute the tasks at the specific time. It does not wait for the completion of previous task. The values should be in milliseconds. The sample code is shown here:

```

@Scheduled(fixedRate = 1000)
public void fixedRateSch() {
}

```

A sample code for executing a task on every second from the application startup is shown here:

```

package com.tutorialspoint.demo.scheduler;

import java.text.SimpleDateFormat;
import java.util.Date;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;

@Component
public class Scheduler {
    @Scheduled(fixedRate = 1000)
    public void fixedRateSch() {
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss.SSS");
    }
}

```



```

        Date now = new Date();
        String strDate = sdf.format(now);
        System.out.println("Fixed Rate scheduler:: " + strDate);
    }
}

```

Observe the following screenshot that shows the application that has started at 09:12:00 and after that every second fixed rate scheduler task has executed.

```

2017-12-06 09:12:00.346 INFO 17592 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2017-12-06 09:12:00.360 INFO 17592 --- [main] com.tutorialspoint.demo.DemoApplication : Started DemoApplication in 7.285 seconds (30
M running for 8.084)
Fixed Rate scheduler:: 2017-12-06 09:12:01.231
Fixed Rate scheduler:: 2017-12-06 09:12:02.230
Fixed Rate scheduler:: 2017-12-06 09:12:03.231
Fixed Rate scheduler:: 2017-12-06 09:12:04.230
Fixed Rate scheduler:: 2017-12-06 09:12:05.230
Fixed Rate scheduler:: 2017-12-06 09:12:06.231
Fixed Rate scheduler:: 2017-12-06 09:12:07.231
Fixed Rate scheduler:: 2017-12-06 09:12:08.231

```

## Fixed Delay

Fixed Delay scheduler is used to execute the tasks at a specific time. It should wait for the previous task completion. The values should be in milliseconds. A sample code is shown here:

```

@Scheduled(fixedDelay = 1000, initialDelay = 1000)
public void fixedDelaySch() {
}

```

Here, the initialDelay is the time after which the task will be executed the first time after the initial delay value.

An example to execute the task for every second after 3 seconds from the application startup has been completed is shown below:

```

package com.tutorialspoint.demo.scheduler;

import java.text.SimpleDateFormat;
import java.util.Date;

import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;

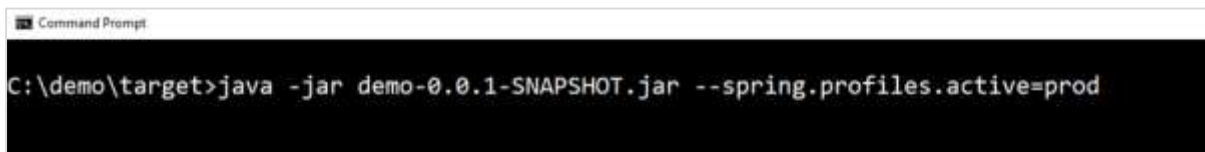
@Component
public class Scheduler {

    @Scheduled(fixedDelay = 1000, initialDelay = 3000)
    public void fixedDelaySch() {

```

```
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd  
HH:mm:ss.SSS");  
  
        Date now = new Date();  
        String strDate = sdf.format(now);  
        System.out.println("Fixed Delay scheduler:: " + strDate);  
    }  
}
```

Observe the following screenshot which shows the application that has started at 09:18:39 and after every 3 seconds, the fixed delay scheduler task has executed on every second.



```
Command Prompt  
C:\demo\target>java -jar demo-0.0.1-SNAPSHOT.jar --spring.profiles.active=prod
```

## 24. Spring Boot – Enabling HTTPS

By default, Spring Boot application uses HTTP 8080 port when the application starts up.

```
2017-11-26 13:56:27.912 INFO 13204 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2017-11-26 13:56:27.922 INFO 13204 --- [main] com.tutorialspoint.demo.DemoApplication : Started DemoApplication in 5.961 seconds (JVM running for 6.933)
```

You need to follow the steps given below to configure the HTTPS and the port 443 in Spring Boot application:

- Obtain the SSL certificate – Create a self-signed certificate or get one from a Certificate Authority
- Enable HTTPS and 443 port

### Self-Signed Certificate

To create a self-signed certificate, Java Run Time environment comes bundled with certificate management utility key tool. This utility tool is used to create a Self-Signed certificate. It is shown in the code given here:

```
keytool -genkey -alias tomcat -storetype PKCS12 -keyalg RSA -keysize 2048 -keystore keystore.p12 -validity 3650
```

Enter keystore password:

Re-enter new password:

What is your first and last name?

[Unknown]:

What is the name of your organizational unit?

[Unknown]:

What is the name of your organization?

[Unknown]:

What is the name of your City or Locality?

[Unknown]:

What is the name of your State or Province?

[Unknown]:

What is the two-letter country code for this unit?

[Unknown]:

Is CN=Unknown, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown correct?

[no]: yes

This code will generate a PKCS12 keystore file named as keystore.p12 and the certificate alias name is tomcat.

## Configure HTTPS

We need to provide the server port as 443, key-store file path, key-store-password, key-store-type and key alias name into the application.properties file. Observe the code given here:

```
server.port: 443
server.ssl.key-store: keystore.p12
server.ssl.key-store-password: springboot
server.ssl.keyStoreType: PKCS12
server.ssl.keyAlias: tomcat
```

You can use the following code if you are using YAML properties use below application.yml:

```
server:
  port: 443
  ssl:
    key-store: keystore.p12
    key-store-password: springboot
    keyStoreType: PKCS12
    keyAlias: tomcat
```

You can create an executable JAR file, and run the spring boot application by using the following Maven or Gradle commands.

For Maven, you can use the following command:

```
mvn clean install
```

After "BUILD SUCCESS", you can find the JAR file under the target directory.

For Gradle, you can use the command

```
gradle clean build
```

After "BUILD SUCCESSFUL", you can find the JAR file under the build/libs directory.

Now, run the JAR file by using the following command:

```
java -jar <JARFILE>
```

Now, the application has started on the Tomcat port 443 with https as shown:

```
2017-12-06 09:49:55.546 INFO 14972 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 443 (https)
2017-12-06 09:49:55.556 INFO 14972 --- [main] com.tutorialspoint.demo.DemoApplication : Started DemoApplication in 9.332 seconds (JVM
```

# 25. Spring Boot – Eureka Server

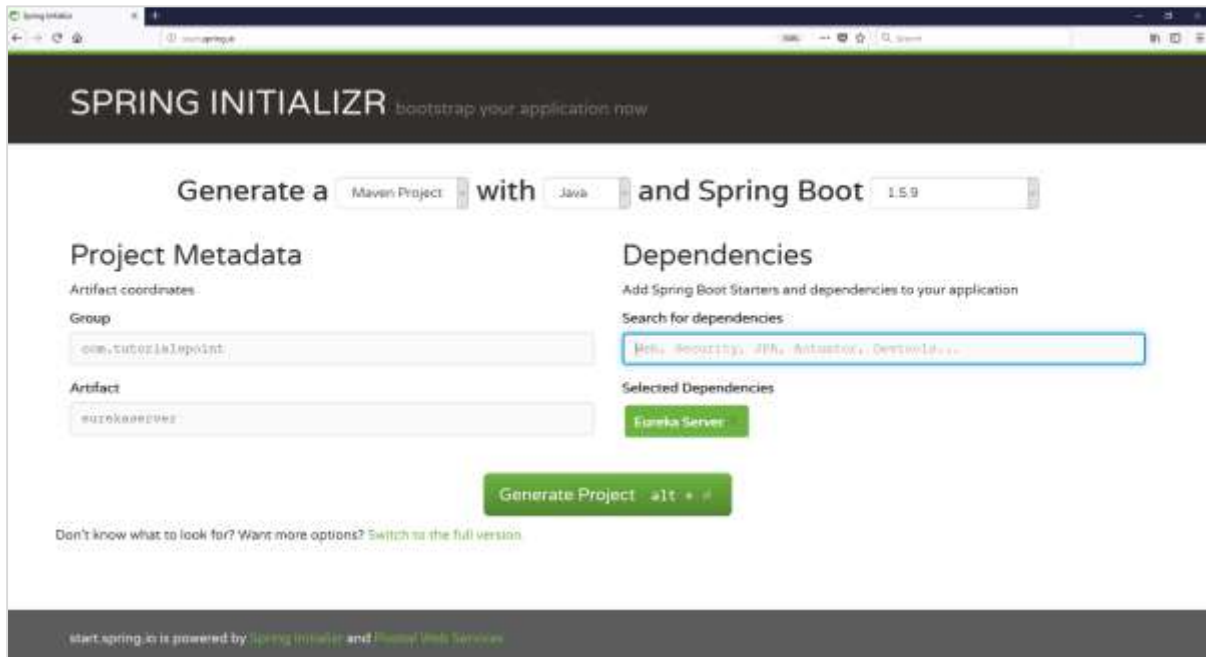
Eureka Server is an application that holds the information about all client-service applications. Every Micro service will register into the Eureka server and Eureka server knows all the client applications running on each port and IP address. Eureka Server is also known as Discovery Server.

In this chapter, we will learn in detail about How to build a Eureka server.

## Building a Eureka Server

Eureka Server comes with the bundle of Spring Cloud. For this, we need to develop the Eureka server and run it on the default port 8761.

Visit the Spring Initializer homepage <http://start.spring.io/> and download the Spring Boot project with Eureka server dependency. It is shown in the screenshot below:



After downloading the project in main Spring Boot Application class file, we need to add `@EnableEurekaServer` annotation. The `@EnableEurekaServer` annotation is used to make your Spring Boot application acts as a Eureka Server.

The code for main Spring Boot application class file is as shown below:

```
package com.tutorialspoint.eureka.server;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;
```

```

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

```

Make sure Spring cloud Eureka server dependency is added in your build configuration file.

The code for Maven user dependency is shown below:

```

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

```

The code for Gradle user dependency is given below:

```

compile('org.springframework.cloud:spring-cloud-starter-eureka-server')

```

The complete build configuration file is given below:

### Maven pom.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>com.tutorialspoint</groupId>
    <artifactId>eureka-server</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>eureka-server</name>
    <description>Demo project for Spring Boot</description>

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

```

```

        <version>1.5.9.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
        <spring-cloud.version>Edgware.RELEASE</spring-cloud.version>
    </properties>

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

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

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>${spring-cloud.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

```

```

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

</project>

```

### Gradle – build.gradle

```

buildscript {
    ext {
        springBootVersion = '1.5.9.RELEASE'
    }
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-
plugin:${springBootVersion}")
    }
}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'org.springframework.boot'

group = 'com.tutorialspoint'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = 1.8

repositories {
    mavenCentral()
}

```



```

}

ext {
    springCloudVersion = 'Edgware.RELEASE'
}

dependencies {
    compile('org.springframework.cloud:spring-cloud-starter-eureka-server')
    testCompile('org.springframework.boot:spring-boot-starter-test')
}

dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-
dependencies:${springCloudVersion}"
    }
}

```

By default, the Eureka Server registers itself into the discovery. You should add the below given configuration into your application.properties file or application.yml file.

The application.properties file is given below:

```

eureka.client.registerWithEureka=false
eureka.client.fetchRegistry=false
server.port=8761

```

The application.yml file is given below:

```

eureka:
  client:
    registerWithEureka: false
    fetchRegistry: false
  server:
    port: 8761

```

Now, you can create an executable JAR file, and run the Spring Boot application by using the Maven or Gradle commands shown below:

For Maven, use the command as shown below:

```
mvn clean install
```

After "BUILD SUCCESS", you can find the JAR file under the target directory.

For Gradle, you can use the command shown below:

```
gradle clean build
```

After "BUILD SUCCESSFUL", you can find the JAR file under the build/libs directory.

Now, run the JAR file by using the following command:

```
java -jar <JARFILE>
```

You can find that the application has started on the Tomcat port 8761 as shown below:

```
2017-12-07 08:59:13.475 INFO 11680 --- [main] S.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8761 (http)
2017-12-07 08:59:13.579 INFO 11680 --- [main] com.e.c.s.EurekaAutoServiceRegistration : Updating port to 8761
2017-12-07 08:59:13.589 INFO 11680 --- [main] e.t.e.EurekaServerApplication : Started eureka-server-application in 24.448 seconds (709 running for 25.861)
2017-12-07 08:59:14.003 INFO 11680 --- [Thread-11] o.s.c.r.n.s.server.EurekaServerBootstrap : isAwake returned false
2017-12-07 08:59:14.008 INFO 11680 --- [Thread-11] o.s.c.r.n.s.server.EurekaServerBootstrap : Initialized server context
2017-12-07 08:59:14.069 INFO 11680 --- [Thread-11] o.n.e.r.FeederAwareInstanceRegistryImpl : Got 1 instances from neighboring DS node
2017-12-07 08:59:14.011 INFO 11680 --- [Thread-11] o.n.e.r.FeederAwareInstanceRegistryImpl : Remove threshold is: 1
2017-12-07 08:59:14.014 INFO 11680 --- [Thread-11] o.n.e.r.FeederAwareInstanceRegistryImpl : Changing status to UP
2017-12-07 08:59:14.034 INFO 11680 --- [Thread-11] e.s.e.EurekaServerInitializerConfiguration : Started Eureka Server
2017-12-07 08:59:00.430 INFO 11680 --- [io-8761-exec-10] o.s.a.c.c.f.[Tomcat].[localhost].[/] : Initializing Spring FrameworkServlet 'dispatcherServlet'
2017-12-07 08:59:00.464 INFO 11680 --- [io-8761-exec-10] o.s.w.s.servlet.DispatcherServlet : FrameworkServlet 'dispatcherServlet': initialization started
2017-12-07 08:59:00.506 INFO 11680 --- [io-8761-exec-10] o.s.w.s.servlet.DispatcherServlet : FrameworkServlet 'dispatcherServlet': initialization completed in 119 ms
```

Now, hit the URL <http://localhost:8761/> in your web browser and you can find the Eureka Server running on the port 8761 as shown below:

The screenshot shows the Spring Eureka web interface in a browser. The page has a dark header with the 'spring Eureka' logo and navigation links for 'HOME' and 'LAST 1000 SINCE STARTUP'. The main content area is divided into several sections:

- System Status:** A table showing environment (S01), data center (Default), current time (2017-12-07T08:51:23+05:30), uptime (00:02), leader election enabled (false), replicas threshold (1), and replicas (Default) (0).
- DS Replicas:** A section showing 'localhost' as the only replica.
- Instances currently registered with Eureka:** A table with columns 'App name', 'Addr', 'Availability Zones', and 'Status'. It shows 'No instances available'.
- General Info:** A table with columns 'Name' and 'Value'. It lists:
  - Host: 192.168.1.100
  - Host-addr: 192.168.1.100
  - Environment: S01
  - Num of peers: 0
  - Current memory usage: 463mb (92%)
  - Server uptime: 00:02
  - Registered replicas: http://localhost:8761/eureka/
  - Unregistered replicas: http://localhost:8761/eureka/
  - Available replicas: http://localhost:8761/eureka/
- Instance Info:** A table with columns 'Name' and 'Value'. It lists:
  - Host: 192.168.1.100
  - Status: UP

## 26. Spring Boot – Service Registration with Eureka

In this chapter, you are going to learn in detail about How to register the Spring Boot Micro service application into the Eureka Server. Before registering the application, please make sure Eureka Server is running on the port 8761 or first build the Eureka Server and run it. For further information on building the Eureka server, you can refer to the previous chapter.

First, you need to add the following dependencies in our build configuration file to register the microservice with the Eureka server.

Maven users can add the following dependencies into the **pom.xml** file:

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

Gradle users can add the following dependencies into the **build.gradle** file:

```
compile('org.springframework.cloud:spring-cloud-starter-eureka')
```

Now, we need to add the `@EnableEurekaClient` annotation in the main Spring Boot application class file. The `@EnableEurekaClient` annotation makes your Spring Boot application act as a Eureka client.

The main Spring Boot application is as given below:

```
package com.tutorialspoint.eurekaclient;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

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

To register the Spring Boot application into Eureka Server we need to add the following configuration in our application.properties file or application.yml file and specify the Eureka Server URL in our configuration.

The code for application.yml file is given below:

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka
    instance:
      preferIpAddress: true
spring:
  application:
    name: eurekaclient
```

The code for application.properties file is given below:

```
eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka
eureka.client.instance.preferIpAddress=true
spring.application.name=eurekaclient
```

Now, add the Rest Endpoint to return String in the main Spring Boot application and the Spring Boot Starter web dependency in build configuration file. Observe the code given below:

```
package com.tutorialspoint.eurekaclient;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
@EnableEurekaClient
@RestController
public class EurekaclientApplication {

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

```

    @RequestMapping(value = "/")
    public String home() {
        return "Eureka Client application";
    }
}

```

The entire configuration file is given below.

### For Maven user - pom.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>com.tutorialspoint</groupId>
    <artifactId>eurekaclient</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>eurekaclient</name>
    <description>Demo project for Spring Boot</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.9.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
        <spring-cloud.version>Edgware.RELEASE</spring-cloud.version>
    </properties>

```

```

</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-eureka</artifactId>
    </dependency>
    <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>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
</project>

```

**For Gradle user – build.gradle**

```
buildscript {
    ext {
        springBootVersion = '1.5.9.RELEASE'
    }
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-
plugin:${springBootVersion}")
    }
}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'org.springframework.boot'

group = 'com.tutorialspoint'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = 1.8

repositories {
    mavenCentral()
}

ext {
    springCloudVersion = 'Edgware.RELEASE'
}

dependencies {
    compile('org.springframework.cloud:spring-cloud-starter-eureka')
    testCompile('org.springframework.boot:spring-boot-starter-test')
    compile('org.springframework.boot:spring-boot-starter-web')
}

dependencyManagement {
```

```

imports {
    mavenBom "org.springframework.cloud:spring-cloud-
dependencies:${springCloudVersion}"
}
}

```

You can create an executable JAR file, and run the Spring Boot application by using the following Maven or Gradle commands:

For Maven, you can use the following command:

```
mvn clean install
```

After "BUILD SUCCESS", you can find the JAR file under the target directory.

For Gradle, you can use the following command:

```
gradle clean build
```

After "BUILD SUCCESSFUL", you can find the JAR file under the **build/libs** directory.

Now, run the JAR file by using the command as shown:

```
java -jar <JARFILE>
```

Now, the application has started on the Tomcat port 8080 and Eureka Client application is registered with the Eureka Server as shown below:

```

2017-12-07 19:16:15.384 INFO 9292 --- [efkclient-0] com.netflix.discovery.DiscoveryClient : DiscoveryClient_EUREKA-1101/ASDFVGHJ-17448 : eurekaclient: registering service...
2017-12-07 19:16:16.148 INFO 9292 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2017-12-07 19:16:16.198 INFO 9292 --- [main] s.b.c.e.s.EurekaWebServiceRegistration : Updating port to 8080
2017-12-07 19:16:16.172 INFO 9292 --- [main] c.t.e.EurekaClientApplication : Started EurekaClientApplication in 12.37 seconds (JVM running for 31.319)
2017-12-07 19:16:16.194 INFO 9292 --- [efkclient-0] com.netflix.discovery.DiscoveryClient : DiscoveryClient_EUREKA-1101/ASDFVGHJ-17448 : eurekaclient - registration status: 200

```

Hit the URL <http://localhost:8761/> in your web browser and you can see the Eureka Client application is registered with Eureka Server.

The screenshot shows the Spring Eureka web interface. At the top, there's a navigation bar with 'HOME' and 'LAST 1000 SINCE STARTUP'. Below this, the 'System Status' section displays a table with system information:

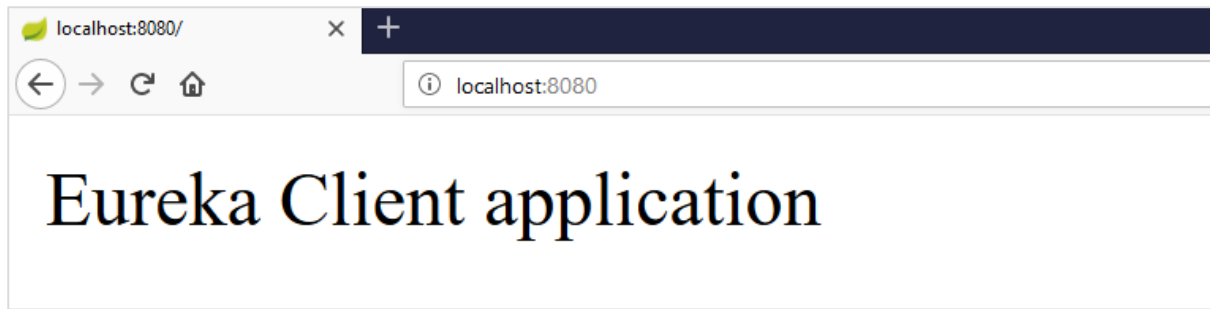
System Status	
Environment	test
Data center	default
Current time	2017-12-07T19:16:59+05:30
Uptime	00:01
Lease expiration enabled	false
Renewal threshold	3
Renewal back-off	0

Below the system status, there's a section for 'Instances currently registered with Eureka'. It shows a table with columns: Application, ARBs, Availability Zones, and Status.

Application	ARBs	Availability Zones	Status
EUREKAClient	1/1	1/1	UP (1) - ASDFVGHJ-17448 - eurekaclient

Now hit the URL <http://localhost:8080/> in your web browser and see the Rest Endpoint output.





# 27. Spring Boot – Zuul Proxy Server and Routing

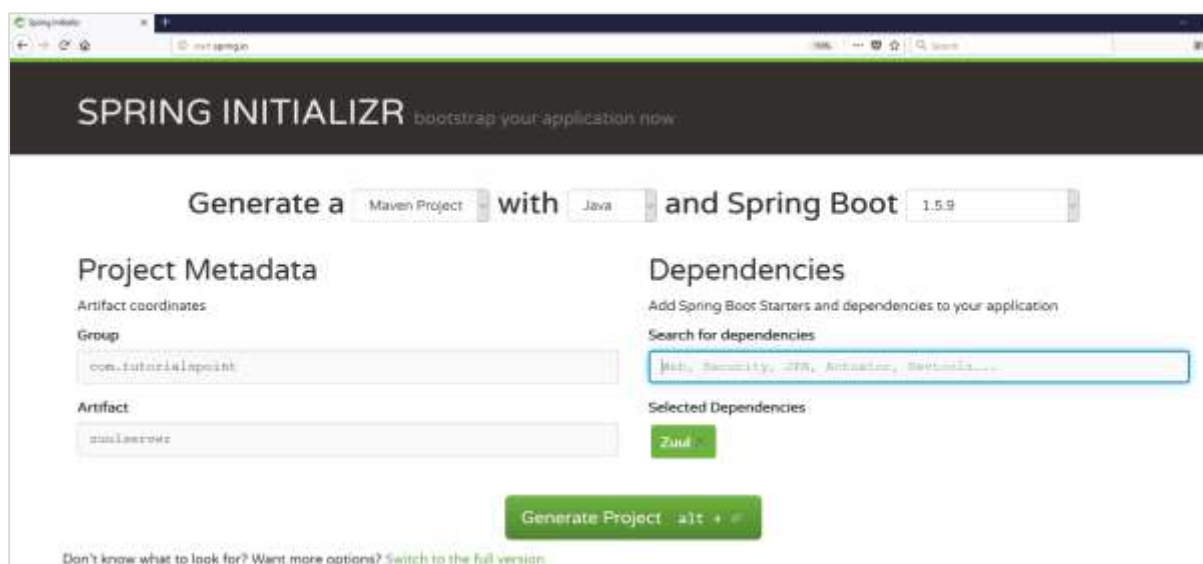
Zuul Server is a gateway application that handles all the requests and does the dynamic routing of microservice applications. The Zuul Server is also known as Edge Server.

For Example, **/api/user** is mapped to the user service and **/api/products** is mapped to the product service and Zuul Server dynamically routes the requests to the respective backend application.

In this chapter, we are going to see in detail how to create Zuul Server application in Spring Boot.

## Creating Zuul Server Application

The Zuul Server is bundled with Spring Cloud dependency. You can download the Spring Boot project from Spring Initializer page <http://start.spring.io/> and choose the Zuul Server dependency.



Add the `@EnableZuulProxy` annotation on your main Spring Boot application. The `@EnableZuulProxy` annotation is used to make your Spring Boot application act as a Zuul Proxy server.

```
package com.tutorialspoint.zuulserver;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.zuul.EnableZuulProxy;

@SpringBootApplication
```

```
@EnableZuulProxy
public class ZuulserverApplication {

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

You will have to add the Spring Cloud Starter Zuul dependency in our build configuration file.

Maven users will have to add the following dependency in your **pom.xml** file:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zuul</artifactId>
</dependency>
```

For Gradle users, add the below dependency in your build.gradle file

```
compile('org.springframework.cloud:spring-cloud-starter-zuul')
```

For Zuul routing, add the below properties in your application.properties file or application.yml file.

```
spring.application.name=zuulserver
zuul.routes.products.path=/api/demo/**
zuul.routes.products.url=http://localhost:8080/
server.port=8111
```

This means that http calls to **/api/demo/** get forwarded to the products service. For example, **/api/demo/products** is forwarded to **/products**.

yaml file users can use the application.yml file shown below:

```
server:
  port: 8111
spring:
  application:
    name: zuulserver
zuul:
```

```

routes:
  products:
    path: /api/demo/**
    url: http://localhost:8080/

```

**Note:** The <http://localhost:8080/> application should already be running before routing via Zuul Proxy.

The complete build configuration file is given below.

Maven users can use the pom.xml file given below:

```

<?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>com.tutorialspoint</groupId>
  <artifactId>zuulserver</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>zuulserver</name>
  <description>Demo project for Spring Boot</description>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.5.9.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
    <java.version>1.8</java.version>
    <spring-cloud.version>Edgware.RELEASE</spring-cloud.version>

```

```

</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-zuul</artifactId>
    </dependency>

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

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

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

</project>

```

Gradle users can use the build.gradle file given below:

```
buildscript {
    ext {
        springBootVersion = '1.5.9.RELEASE'
    }
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-
plugin:${springBootVersion}")
    }
}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'org.springframework.boot'
group = 'com.tutorialspoint'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = 1.8
repositories {
    mavenCentral()
}
ext {
    springCloudVersion = 'Edgware.RELEASE'
}
dependencies {
    compile('org.springframework.cloud:spring-cloud-starter-zuul')
    testCompile('org.springframework.boot:spring-boot-starter-test')
}
dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-
dependencies:${springCloudVersion}"
    }
}
```

You can create an executable JAR file, and run the Spring Boot application by using the Maven or Gradle commands given below:

For Maven, you can use the command given below:

```
mvn clean install
```

After "BUILD SUCCESS", you can find the JAR file under the target directory.

For Gradle, you can use the command given below:

```
gradle clean build
```

After "BUILD SUCCESSFUL", you can find the JAR file under the build/libs directory.

Now, run the JAR file by using the command shown below:

```
java -jar <JARFILE>
```

You can find the application has started on the Tomcat port 8111 as shown here.

```
2017-12-08 08:09:27.842 INFO 3312 --- [main] ratonshystrisMetricsPollerConfiguration : Starting poller
2017-12-08 08:09:27.957 INFO 3312 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8111 (http)
2017-12-08 08:09:27.968 INFO 3312 --- [main] c.t.haulserver.HaulserverApplication : Started HaulserverApplication in 10.134 seconds (Tom running for 31.125)
```

Now, hit the URL <http://localhost:8111/api/demo/products> in your web browser and you can see the output of **/products** REST Endpoint as shown below:

```
localhost:8111/api/demo/products

// 20171208081158
// http://localhost:8111/api/demo/products

[
  {
    "id": "1",
    "name": "Honey"
  },
  {
    "id": "2",
    "name": "Almond"
  }
]
```