# Spring Boot – Introduction Tutorial

Spring Boot is a framework developed on top of core spring framework.

The main aim of Spring Boot is to let developers to create spring **production** grade applications and services with very less effort.

**Did we remember what it takes to create real-time spring applications?**

* It includes writing many XML configurations, **server** setting, adding **dependencies**…etc. But with spring Boot we can avoid all these boilerplate code, writing XML configurations and annotations.
* We can create real-time production ready applications within minutes.
* Spring Boot comes with inbuilt server, we no longer have to use any external servers like ***Tomcat*, *Glass-fish*** or anything else, so don’t need to deploy WAR files.

**Advantage/Disadvantage**

So, the main advantage of Spring Boot is, we can create spring based applications easily in very less time, without need of any XML configurations.

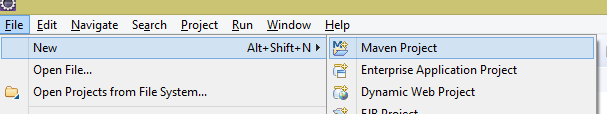
The main disadvantage is, it will be little tough to migrate existing spring enterprise applications to Spring Boot.

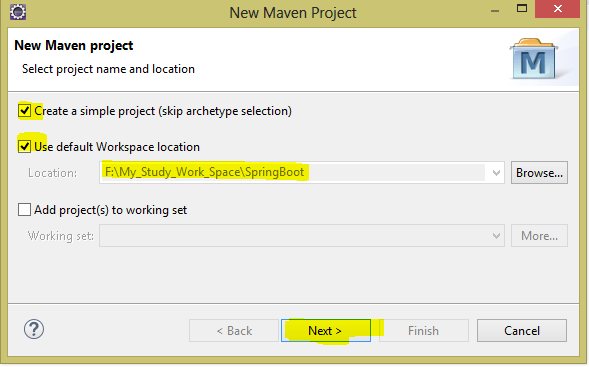
We have to use either **Maven** or **Gradle** **build tool** to work with Spring Boot.

Spring Boot provides command line interface tool to develop/test the Spring Boot applications from the command prompt easily.

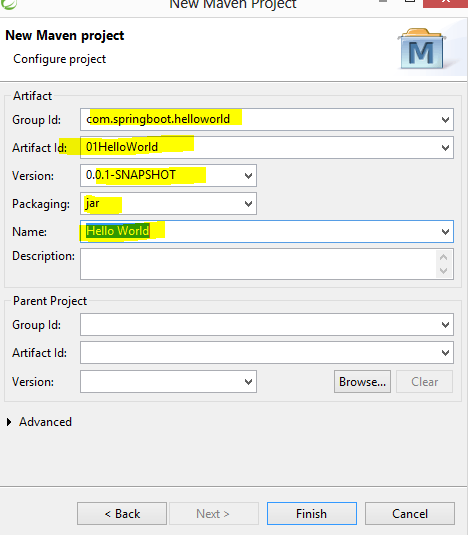
***Spring Boot + Maven – Hello World Example Step by Step***

**1. Open Eclipse > File > New > Maven Project**

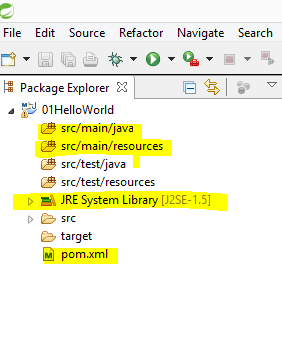
**2. Tick ‘Create a simple project (skip archetype selection) ‘check box > click Next**



**3. Provide Group Id (it’s your package), Artifact Id (project name) and click Finish**



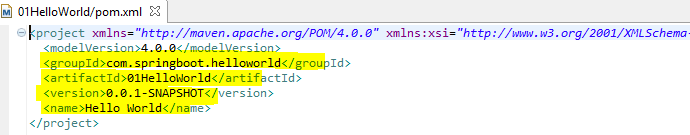
**4. Now you will see a Maven project in your work space, something like..**



**Note**: If we observe, its showing*‘JRE System Library [****J2SE-1.5****]’* as default java version, let’s keep it aside for now.

**5. So, Maven project is created with default setup, lets add Spring Boot related stuff in the pom.xml (in pom.xml we will include all dependencies), open pom.xml**

By default pom.xml contains



**<parent>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-starter-parent</artifactId>**

**<version>1.5.6.RELEASE</version>**

**</parent>**

**<dependencies>**

**<dependency>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-starter-web</artifactId>**

**</dependency>**

**</dependencies>**

**<properties>**

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

**</properties>**

## Explanation

* We have added **spring-boot-starter-parent, spring-boot-starter-web** and java version 8.
* **What is spring-boot-starter-parent?**

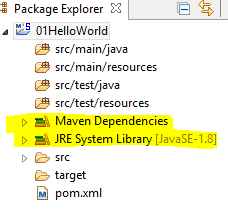
🡪 Actually this is an existing project given by spring team which contains Spring Boot supporting configuration data (remember just configuration data, it won’t download any jars).

🡪 We have added this in a **<parent>** tag means, we are instructing Maven to consider our **01BootHelloWorld** project is the child project of it.

* In the dependencies, I have added spring-boot-starter-web for web module

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

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



**<dependency>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-starter-web</artifactId>**

**</dependency>**

**Note**:

In the above dependency I haven’t included version number for **spring-boot-starter-web.**

But maven downloaded some jar files with some version(s) related to **spring-boot-starter-web**, how it’s possible?

That’s because of Maven’s parent child relation.

While adding spring boot parent project, I have included version as **1.5.6.RELEASE**, so again **we no need to add version numbers for the dependencies**.

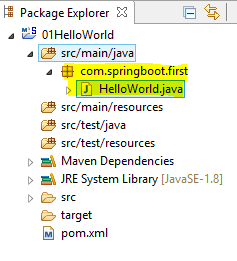
As told earlier, **spring-boot-starter-parent** contains configuration Meta data, this means.

It knows which version of dependency need to be downloaded.  So we no need to worry about dependencies versions. Which will save lot of our time?

**7.  Now create a java class in src/main/java >**

I have created one with name HelloWorld.java in com.springboot.first package.

I mean the final directory structure looks like…



**Note**: put your java class in some package, it is mandatory.\* If you haven’t created a package it gives the following error while running your Spring Boot application.

Our ApplicationContext is unlikely to start due to a @ComponentScan of the default package.

**package** com.springboot.first;

**import** org.springframework.boot.SpringApplication;

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

@SpringBootApplication

**public** **class** HelloWorld {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(HelloWorld.**class**, args);

System.***out***.println("Hello World!!");

}

}

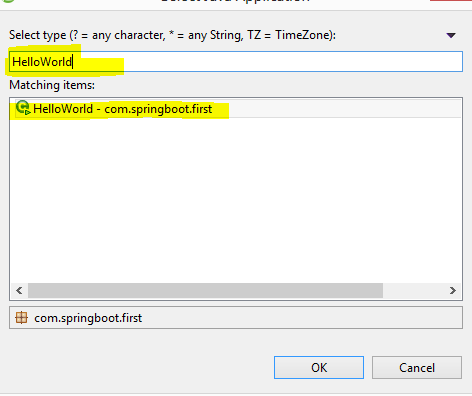
## Explanation

* added @SpringBootApplication annotation, means this is the starting point for our Spring Boot application
* SpringApplication.*run*(HelloWorld.**class**, args)- I am bootstrapping the application

For now just remember, for every spring boot application we have to create a main class and that need to be annotate with @SpringBootApplication and bootstrap it .

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

**Chose your class to run like below**

****

**10. Open console and check the output**

**2019-02-08 00:34:05.457 INFO 6036 --- [ main] o.s.j.e.a.AnnotationMBeanExporter : Registering beans for JMX exposure on startup**

**2019-02-08 00:34:05.581 INFO 6036 --- [ main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 2019 (http)**

**2019-02-08 00:34:05.591 INFO 6036 --- [ main] com.springboot.first.HelloWorld : Started HelloWorld in 7.078 seconds (JVM running for 7.946)**

**Hello World!!**

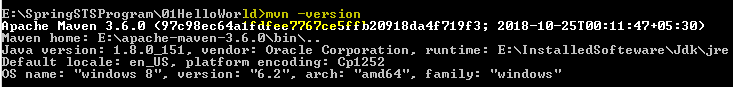
**Or**

***Maven tutorial***

**Install the maven**

The installation of Apache Maven is a simple process of extracting the archive and adding the bin folder with the mvn command to the PATH.

Detailed steps are:

* Ensure JAVA\_HOME environment variable is set and points to your JDK installation
* Extract distribution archive in any directory
* Add the bin directory of the created directory apache-maven-3.6.0 to the PATH environment variable
* Confirm with mvn -v in a new shell. The result should look similar to
* 

**echo %JAVA\_HOME%**

**C:\Program Files\Java\jdk1.7.0\_51**

## Build Spring Boot Project with Maven

To be able to run our Spring Boot we need to first build it.

To build and package a Spring Boot app into a single executable Jar file with a Maven, use the below command

We need to run it from the project folder which contains the pom.xml file.

1. **maven package**

Or

We can also use

1. **mvn install**

## Run Spring Boot app with java -jar command

To run our Spring Boot app from a command line in a Terminal window we can you the **java -jar** command.

This is provided our Spring Boot app was packaged as an executable jar file.

## E:\SpringSTSProgram\01HelloWorld>java -jar target/01HelloWorld-0.0.1-SNAPSHOT.jar

## Run Spring Boot app using Maven

We can also use Maven plug-in to run our Spring Boot app.

Use the below example to run our Spring Boot app with Maven plugin:

1. **mvn spring-boot:run**

**Maven clean – it will remove the old jar.**

**Maven clean install - it will remove and create the new jar**

**Run the application.**

## E:\SpringSTSProgram\01HelloWorld>java -jar target/01HelloWorld-0.0.1-SNAPSHOT.jar

**09. Output:**

**2019-02-08 00:01:01.599 INFO 2308 --- [ main] o.s.j.e.a.AnnotationMBeanExporter : Registering beans for JMX exposure on startup**

**2019-02-08 00:01:01.744 INFO 2308 --- [ main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 2019 (http)**

**2019-02-08 00:01:01.754 INFO 2308 --- [ main] com.springboot.first.HelloWorld : Started HelloWorld in 10.581 seconds (JVM running for 12.088)**

**Hello World!!;**

***Spring Boot – Creating a Restful Web Service Example***

**package** com.springboot;

@SpringBootApplication

**public** **class** SpringBootAppl {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(SpringBootAppl.**class**, args);

}

}

* The class which having annotated with SpringBootApplication should be in the base package or root of the package else while accessing it from the browser it will give the below error message.

White label Error Page This application has no explicit mapping for /error, so you are seeing this as a fallback. Tue Jun 30 17:24:02 CST 2015 There was an unexpected error (type=Not Found, status=404). No message available.

The @SpringBootApplication annotation is equivalent to using @Configuration, @EnableAutoConfiguration and @ComponentScan with their default attributes.

package com.springboot.controller;

@RestController

public class SpringBootController {

@RequestMapping("/")

public String welcome(){

return "welcome to spring boot tutorial!!!";

}

@RequestMapping("/hello")

public String sayHello(){

return "say Hello !! how are you!!";

}

@RequestMapping("/sum")

public int addition(){

return 100+200;

}

}

Application.properties

server.port=2019

***My rest class is annotated with @RestController, which tells Spring Boot to consider this class as REST controlle*r *and register @RequestMapping paths inside it to respond to the HTTP requests*.**

We can check the output by using below command

Now try hitting <http://localhost:2019/> - **welcome to spring boot tutorial!!!**

<http://localhost:2019/hello> - **say Hello!! How are you!!!**

[http://localhost:2019/sum - 300](http://localhost:2019/sum%20-%20300)

***Note***:

I have created Spring Boot main application class in **package** com.springboot and controller class in **package com.springboot.controller***, a*nd in my controller class I have written my Restful service logic and was able to execute the application successfully.

**How spring boot knows to scan our controller?** As we have created our main class in **package** com.springboot, while starting our application, it will scan all the components under that package.

As we have created our controller class in **package com.springboot.controller which** is inside **package com.springboot**, our controller was registered by spring boot.

If we create the controller class outside of the main package, let’s say **package com.spring.controller***,*if we run the application it gives 404 errors,

**What’s the solution for this?**

We have to add *@ComponentScan* annotation in our Spring Boot main class, something like this.

@SpringBootApplication

**@ComponentScan(basePackages="com.java4s.controller")**

public class SpringBootApp {

public static void main(String[] args){

---

}

}

**Common Application Properties (application.properties)**

Generally we will create property files for writing static values related to our application.

If we consider some real time java applications, we will use these .property files for writing environmental (server) related stuff and for even for some static content those are suppose to change frequently.

By default, Spring Boot provides a .properties file with name ***application.properties*** with some predefined key’s in it

Path: /src/main/resources

How to Change Default Tomcat Server Port

In our previous [Restful](https://www.java4s.com/spring-boot-tutorials/spring-boot-creating-a-restful-web-service-example/) example, when we start the application Spring Boot’s inbuilt tomcat server by default will take 8080 as its port number.

In Spring Boot, we can change tomcat’s port number in 2 ways…

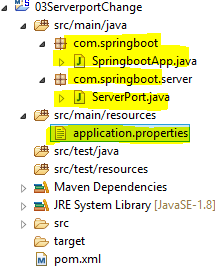
* **Using application.properties**
* Using Java code change

Firstly, let me show you using ***application.properties.***

I just **created application.properties file in *src/main/resources*** no other changes.

***Application.properites***

**server.port=2020**

****

## Using Java code change

In this approach, we will create a simple java class which implements **EmbeddedServletContainerCustomizer** interface of Spring Boot.

This is a strategy interface for customizing **auto-configured** embedded servlet containers, and we need to override **customize** () method of that interface as below.

**package com.springboot.server;**

import org.springframework.boot.context.embedded.ConfigurableEmbeddedServletContainer;

import org.springframework.boot.context.embedded.EmbeddedServletContainerCustomizer;

import org.springframework.stereotype.Component;

**@Component**

**public class ServerPort implements EmbeddedServletContainerCustomizer {**

**@Override**

**public void customize(ConfigurableEmbeddedServletContainer container) {**

**container.setPort(2040);**

**}**

**}**

**Below code is the common for all the spring boot application**

@SpringBootApplication

**public** **class** SpringbootApp {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(SpringbootApp.**class**, args);

}

}

**Note**:

If we use both application.properties and Java config, Spring Boot will give preference for Java only, I mean it takes 2040 as its tomcat port number

Console output.

2019-02-08 06:46:51.056 INFO 5524 --- [ main] o.s.j.e.a.AnnotationMBeanExporter : Registering beans for JMX exposure on startup

2019-02-08 06:46:51.142 INFO 5524 --- [ main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): **2040** (http)

2019-02-08 06:46:51.149 INFO 5524 --- [ main] com.springboot.SpringbootApp : Started SpringbootApp in 6.287 seconds (JVM running for 7.128)

***Spring Boot – How to Change Default Context Path***

Firstly what is this context path? Simply it’s our application name.

Generally while we are hitting any application in the browser, we will write the URL with the application name (context) right?

http://localhost:<port>/***<***application\_name or context\_path***>***/operation\_name

But if we check earlier example we haven’t included any context path, we directly ran the application with the path we have given in @RequestMapping.

Spring Boot by default considers the context path as ‘**/**‘so we no need to give our application name or context path, but in real-time we should use some context path for the applications.

How to change default spring boot application context path ‘/‘to our application name?

In Spring Boot, we can change application default context path in two ways.

* Using applications.properties
* Using Java code changes

It’s very simple just like changing tomcat port number.

## Using application.properties

Create application.properties in your application src/main/resources and write this line.

**server.contextPath=/contextPath**

@SpringBootApplication

**public** **class** ContextPath {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(ContextPath.**class**, args);

System.***out***.println("Application has started.");

}

}

@RestController

**public** **class** Mapping {

@RequestMapping("/context")

**public** String changeContextPath(){

**return** "Application context file got changed!!!";

} }

@Component

**public** **class** ChangeContextPath **implements** EmbeddedServletContainerCustomizer {

@Override

**public** **void** customize(ConfigurableEmbeddedServletContainer container) {

//To change the context path

container.setContextPath("/contextPath");

container.setPort(2025);

} }

Now we have to run the application by hitting [http://localhost:2025/*yourApplicationName*/](http://localhost:2025/yourApplicationName/),

**http://** [**http://localhost:2025/contextPath/context**](http://localhost:2025/contextPath/context)

**Output**: Application context file got changed!!!

**Note**: If we use both java and properties file approaches, spring boot will consider java only.

How to Reload Changes without Restarting the Server

One of the main challenges for the java developers is to deploy the apps and restart server whenever there is a code change.

How to reload the code changes without having to restart the server.

In Spring Boot this can be achieved by adding a **DevTools** module, just add the following dependency in your Spring Boots pom.xml and build it.

**<dependencies>**

**<dependency>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-starter-web</artifactId>**

**</dependency>**

**<dependency>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-devtools</artifactId>**

**</dependency>**

**</dependencies>**

Spring Boot DevTools module does exactly what developers needed; this eliminates the process of manually deploying the changes.

DevTools will auto restart the server when we have changes.

Spring team they haven’t included this feature in Spring Boot’s initial version, upon several requests they added this feature later.

We can start the server and run the application.

Now if we do some code changes to your project files, no need to restart the server again to view the changes rather just refresh your browser.

***Spring Boot JDBC + Oracle – How to Create/Configure a Data Source***

**How to earlier we are connecting by using java?**

**package** com.connect;

**import** java.sql.Connection;

**import** java.sql.DriverManager;

**import** java.sql.PreparedStatement;

**import** java.sql.SQLException;

**import** java.sql.Statement;

**public** **class** DataBaseConnectivity {

**static** String *QUERY* ="CREATE TABLE Employee (firstName VARCHAR(20),lastName VARCHAR(20),age INT,employeeID INT NOT NULL)";

**public** **static** Connection getConnection() {

Connection con = **null**;

PreparedStatement prstmt = **null**;

**try** {

Class.*forName*("oracle.jdbc.driver.OracleDriver");

con = DriverManager.*getConnection*(

"jdbc:oracle:thin:@localhost:1521:XE", "system", "amit");

} **catch** (Exception e) {

e.printStackTrace();

}

**return** con;

}

**public** **static** **void** main(String[] args) {

Connection connection = *getConnection*();

**try** {

Statement statement = connection.createStatement();

//Table Creation

/\*boolean flag = statement.execute(QUERY);

System.out.println("TABLE Employee has been created!! “);\*/

//Insert method

*insert*(statement);

} **catch** (SQLException e) {

e.printStackTrace();

}

}

**public** **static** **void** insert(Statement stmt){

**try** {

stmt.executeUpdate("INSERT INTO Employee " + "VALUES ('Amit', 'Kumar', 29, 46978)");

System.***out***.println("Records has been inserted successfully");

} **catch** (SQLException e) {

e.printStackTrace();

} **finally** {

**try** {

**if** (stmt != **null**) {

stmt.close();

}

} **catch** (Exception e) {

e.printStackTrace();

}

}

}

}

We are all aware that the process of creating a traditional Spring JDBC application is little tedious because of its XML configurations and in spring boot we need to follow few steps to configure any data source.

## Steps to Create Data Source in Spring Boot Application

* Add Spring Boot JDBC dependency in **POM**.xml
* Add data source information in application.properties
* Get **JDBCTemplate** object in your DAO with @**Autowired** annotation and use it

## 1. Add Spring Boot JDBC Dependency

<!-- Spring boot jdbc dependency -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-jdbc</artifactId>

</dependency>

**And**

// for Oracle Driver

<dependency>

<groupId>com.oracle</groupId>

<artifactId>ojdbc6</artifactId>

<version>11.2.0</version>

</dependency>

**Note**: it’s giving the error message as below

[**Missing artifact com.oracle:ojdbc6: jar: 11.2.0,**](https://stackoverflow.com/questions/44238622/missing-artifact-com-oracleojdbc6jar11-2-0-3)

Reasons: Unfortunately due the binary license there is no public repository

With the Oracle Driver JAR, so we cannot just add it our pom file.

We have to add this jar manually into the dot m2 repository by using below command.

mvn install:install-file -Dfile=E:\SpringSTSProgram\Ojdbc\_jar\ojdbc6.jar -DgroupId=com.oracle -DartifactId=ojdbc6 -Dversion=11.2.0 -Dpackaging=jar

## 2. Add Datasource Information in application.properties

# Applicationn context name

server.contextPath=/dataSource

# Oracle data source details

spring.datasource.url=jdbc:oracle:thin:@localhost:1521:XE

spring.datasource.username=system

spring.datasource.password=amit

spring.datasource.driver-class-name=oracle.jdbc.driver.OracleDriver

With this we have configured the data source in our spring boot application.

## 3. Get JDBCTemplate object in your DAO with @Autowired annotation

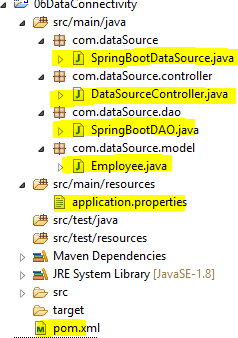
By using @Autowired annotation it will create the JDBCTemplate object at our required DAO classes.

Spring Boot will automatically get the datasource details from application.propeties and injects to jdbcTemplate object while auto wiring.

**@Autowired**

**private JdbcTemplate jdbcTemplate;**

**Directory structure**

****

@SpringBootApplication

public class SpringBootDataSource {

public static void main(String[] args) {

SpringApplication.*run*(SpringBootDataSource.class, args);

System.*out*.println("Server has been started successfyllt!!");

}

}

public class Employee {

private String firstName;

private String lastName;

private BigDecimal age;

private BigDecimal empId;

public Employee() {

System.*out*.println("Emloyee Constructor");

}

public Employee(String firstName, String lastName, BigDecimal age, BigDecimal empId) {

this.firstName = firstName;

this.lastName = lastName;

this.age = age;

this.empId = empId;

}

// Developer the setter and getter method

}

**Properties file**

server.port=2085

# Applicationn context name

server.contextPath=/dataSource

# Oracle data source details

spring.datasource.url=jdbc:oracle:thin:@localhost:1521:XE

spring.datasource.username=system

spring.datasource.password=amit

spring.datasource.driver-class-name=oracle.jdbc.driver.OracleDriver

@Repository

public class SpringBootDAO {

@Autowired

private JdbcTemplate jdbcTemplate;

private static String *QUERY* = "select \* from Employee";

public List<Employee> getEmployeeDetails(){

List<Employee> list = new ArrayList<Employee>();

List<Map<String,Object>> mapList=jdbcTemplate.queryForList(*QUERY*);

for (Map<String, Object> map : mapList) {

Employee emp = new Employee();

emp.setFirstName((String)map.get("FIRSTNAME"));

emp.setLastName((String)map.get("LASTNAME"));

emp.setAge((BigDecimal)map.get("AGE"));

emp.setEmpId((BigDecimal)map.get("EMPLOYEEID"));

list.add(emp);

}

return list;

}

}

@RestController

public class DataSourceController {

@Autowired

private SpringBootDAO dao;

@RequestMapping("/employee")

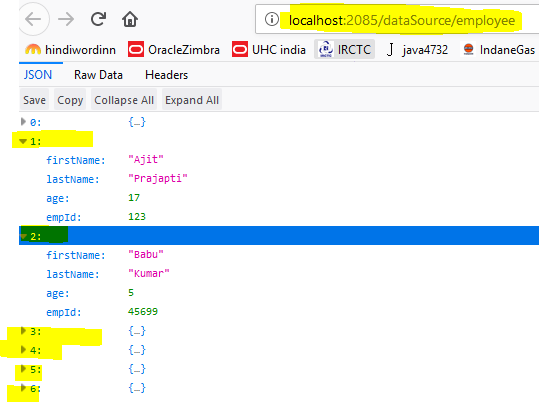
public List<Employee> getEmployeeInfo(){

return dao.getEmployeeDetails();

}

}

**Output**



# How to Configure Multiple Data Source

In order to create multiple data sources we may need to write little configuration.

Need to add one more dependency as below

**<dependency>**

**<groupId>com.oracle</groupId>**

**<artifactId>ojdbc6</artifactId>**

**<version>11.2.0</version>**

**</dependency>**

**<!-- Spring boot jdbc dependency -->**

**<dependency>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-starter-jdbc</artifactId>**

**</dependency>**

**<!-- For multi data source-->**

**<dependency>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-configuration-processor</artifactId>**

**</dependency>**

**Application.properties**

server.port=2085

# Applicationn context name

server.contextPath=/multidatasource

# Oracle data source details

spring.datasource.url=jdbc:oracle:thin:@localhost:1521:XE

spring.datasource.username=system

spring.datasource.password=amit

spring.datasource.driver-class-name=oracle.jdbc.driver.OracleDriver

# 2nd Oracle data source

spring.datasource.second.url=jdbc:oracle:thin:@localhost:1521:XE

spring.datasource.second.username=system

spring.datasource.second.password=amit

spring.datasource.second.driver-class-name=oracle.jdbc.driver.OracleDriver

**Configuration**

**package** com.second.ds.config;

@Configuration

**public** **class** MutiDataSourceConfig {

@Bean

@Primary

@ConfigurationProperties(prefix="spring.datasource")

**public** DataSource getFirstDataSource(){

**return** DataSourceBuilder.*create*().build();

}

----------------------

@Bean

@ConfigurationProperties(prefix="spring.datasource.second")

**public** DataSource getSecondDataSource(){

**return** DataSourceBuilder.*create*().build();

}

--------------------

//getFirstDataSource Should be the same as above getFirstDataSource()

//by Using jdbcTemplateFirst we can use it in the DAO class

@Bean

**public** JdbcTemplate

jdbcTemplateFirst(@Qualifier("getFirstDataSource") DataSource firstDs)

{

**return** **new** JdbcTemplate(firstDs);

}

--------------------

//getSecondDataSource Should be the same as above getSecondDataSource()

//by Using jdbcTemplateSecond we can use it in the DAO class

@Bean

**public** JdbcTemplate jdbcTemplateSecond(@Qualifier("getSecondDataSource") DataSource secondDs) {

**return** **new** JdbcTemplate(secondDs);

}

}

@Repository

**public** **class** MultiDataSourceDAO {

@Autowired

**public** JdbcTemplate jdbcTemplateFirst;

@Autowired

**public** JdbcTemplate jdbcTemplateSecond;

**public** List<Employee> getAllEmployeeDetails(){

String query = "SELECT \* FROM Employee";

List<Employee> empList = **new** ArrayList<Employee>();

List<Map<String, Object>> listMap=jdbcTemplateFirst.queryForList(query);

Employee emp = **null**;

**for** (Map<String, Object> map : listMap) {

emp = **new** Employee();

emp.setFirstName((String)map.get("FIRSTNAME"));

emp.setLastName((String)map.get("LASTNAME"));

emp.setAge((BigDecimal)map.get("AGE"));

emp.setEmpId((BigDecimal)map.get("EMPLOYEEID"));

empList.add(emp);

}

**return** empList;

}

**public** String secondDataSourceValidation(){

String query = "SELECT COUNT(\*) FROM DUAL";

String output=jdbcTemplateSecond.queryForObject(query, String.**class**);

String status="";

**if**(output.equals(1)){

status = "Second Data Source Connection Successfully!!";

}**else**{

status="Second Data Source Connection not Successfull!!!!";

}

**return** status;

}

}

@SpringBootApplication

**public** **class** SpringBootMultiDataSource {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(SpringBootMultiDataSource.**class**, args);

System.***out***.println("Server has been started successfully !!!!");

}

}

@RestController

**public** **class** SecondDataSourceController {

@Autowired

**private** MultiDataSourceDAO dao;

@RequestMapping("/empDetails")

**public** List<Employee> getEmployeeDetailsWithfirtDataSource(){

**return** dao.getAllEmployeeDetails();

}

@RequestMapping("/secondDSValidation")

**public** String secondDSValidation(){

**return** dao.secondDataSourceValidation();

}

}

**public** **class** Employee {

**private** String firstName;

**private** String lastName;

**private** BigDecimal age;

**private** BigDecimal empId;

**public** Employee() {

System.***out***.println("Emloyee Constructor");

}

**public** Employee(String firstName, String lastName, BigDecimal age,

BigDecimal empId) {

**this**.firstName = firstName;

**this**.lastName = lastName;

**this**.age = age;

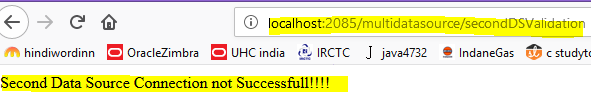
**this**.empId = empId;

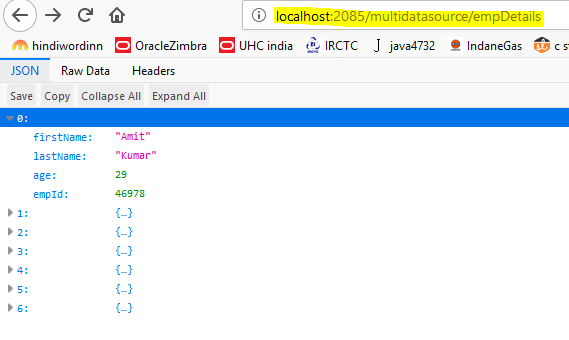
}

// develop setter and getter method

}

Output

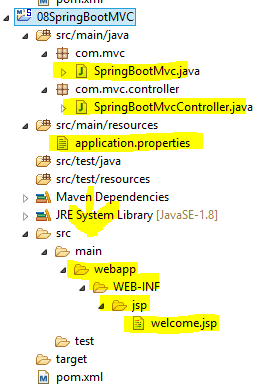




***Spring Boot + Spring MVC + JSP Hello World Example***

We have to create a webapp folder under **/src/main (src > main > webapp)** where we will place all our .jsp files.

Directory structure



## Dependencies

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<!-- For server auto start -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

</dependency>

<**dependency>**

**<groupId>org.apache.tomcat.embed</groupId>**

**<artifactId>tomcat-embed-jasper</artifactId>**

**<scope>provided</scope>**

**</dependency>**

</dependencies>

## application.properties

# Applicationn context name

server.contextPath=/springbootmvc

server.port=2085

# Spring MVC related

spring.mvc.view.prefix=/WEB-INF/jsp/

spring.mvc.view.suffix=.jsp

All .jsp file(s) under /WEB-INF/jsp/ folder, no need to mention ‘webapp’ in the properties file as spring boot by default consider that folder and search under webapp.

@SpringBootApplication

**public** **class** SpringBootMvc {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(SpringBootMvc.**class**, args);

System.***out***.println("Server has started successfully!!!");

}

}

@RestController

**public** **class** SpringBootMvcController {

@GetMapping("/")

**public** ModelAndView showLoginPage(Model model) {

model.addAttribute("message", "You are welcom");

**return** **new** ModelAndView("welcome");

}

}

Welcome.jsp

<%@ page language=*"java"* contentType=*"text/html; charset=ISO-8859-1"*

pageEncoding=*"ISO-8859-1"*%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=ISO-8859-1"*>

<title>Insert title here</title>

</head>

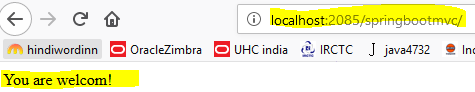
<body>

${message}!

</body>

</html>

**Output**



*Spring Boot – Example of Restful Web Service with XML Response - VVI*

Spring boot services by default gives the response in JSON format, but we can reverse this functionality in such a way that the default response will be in XML.

In order to do that we have to add a new dependency called

**jackson-dataformat-xml.**

With this dependency services by default gives the response in XML format and if we want to see the response in JSON, just append .json to the URL.

**<!-- xml dependency -->**

**<dependency>**

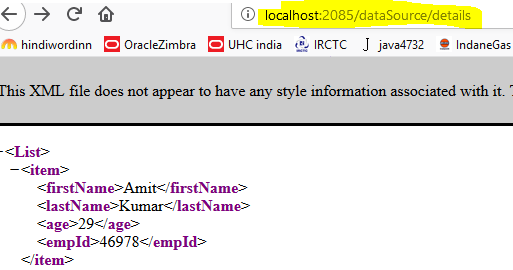
**<groupId>com.fasterxml.jackson.dataformat</groupId>**

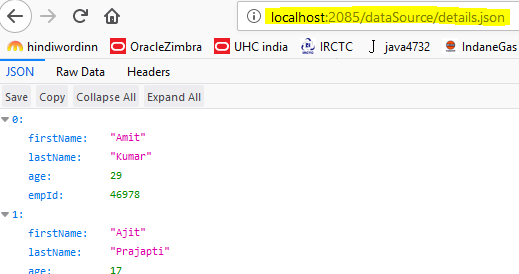
**<artifactId>jackson-dataformat-xml</artifactId>**

**</dependency>**

**<!-- xml dependency end -->**

Output





*Display All Beans Available in Application Context*

How to see the beans that are loaded by the Spring Boot from the Application Context

What we have to do is implement main class with **CommandLineRunner/ApplicationRunner** interface and override its run method.

CommandLineRunner/ApplicationRunner’s run() method will get execute right after ApplicationContext is created and before Spring Boot application initialized.

This run() method will execute only once in an application’s life cycle.

So what’s the difference between **CommandLineRunner/ApplicationRunner?** Basically, both will do the same operation,

The only difference is CommandLineRunner’s run() method will accept String array and ApplicationRunner’s run() will accept ApplicationArguments as arguments.

@SpringBootApplication

**public** **class** SpringBootBeanObjectDisplay **implements** **ApplicationRunner** {

@Autowired

**private** ApplicationContext context;

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(SpringBootBeanObjectDisplay.**class**, args);

System.***out***.println("Server has started successfully!!!!!!!!!!!!!");

}

@Override

**public** **void** run(ApplicationArguments arg0) **throws** Exception {

String[] beans = context.getBeanDefinitionNames();

**for** (String beanName: beans) {

System.***out***.println(beanName);

}

}

// Implements the interface CommandLineRunner

/\*public void run(String... arg0) throws Exception {

String[] beans = context.getBeanDefinitionNames();

for (String beanName: beans) {

System.out.println(beanName);

}

}\*/

}

*How to Configure Cache in Spring Boot Applications*

Caching helps to increase the performance of the application by reducing number of round trips between the database and any expensive resources

In real time we will face the scenarios like we have to execute heavy database query and let’s say the data in the database will change very rarely, for this kind of scenarios it’s not a good idea to hit the database for every call, rather just cache the result at the first time when it hits the database and return the same data again for the other calls.

Steps to configure cache in spring boot applications...

* In pom.xml add spring cache dependency ***spring-boot-starter-cache module***
* Enable cache in spring boot application by writing the *@****EnableCaching*** annotation to the main class
* Add @**Cacheable** annotation to the method which we would like to cache the result

<!-- For Caching the frequently used method -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-cache</artifactId>

</dependency>

@SpringBootApplication

**@EnableCaching**

**public** **class** SpringBootCacheEnabling {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(SpringBootCacheEnabling.**class**, args);

System.***out***.println("Server has started successfully!!!!!");

}

}

@RestController

**public** **class** CacheController {

@Autowired

**private** CacheDAO dao;

@RequestMapping("/empDeatils")

**@Cacheable(value="empInfo")**

**public** List<Employee> getEmployeeDetails(){

**return** dao.getEmployeeDetails();

}

}

Application.properties

server.port=2085

# Applicationn context name

server.contextPath=/sharedCache

# Oracle data source details

spring.datasource.url=jdbc:oracle:thin:@localhost:1521:XE

spring.datasource.username=system

spring.datasource.password=amit

spring.datasource.driver-class-name=oracle.jdbc.driver.OracleDriver

url : <http://localhost:2085/sharedCache/empDeatils>

I hit/refreshed the above URL several times and still its showing

Emloyee Constructor

Emloyee Constructor

Emloyee Constructor

Emloyee Constructor

Emloyee Constructor

Emloyee Constructor

Emloyee Constructor

Its try to fetch the seven Employee details from the data base so 7 times Employee Constructor has executed

After hit/refreshed – it will not hit the data base again due to cache management and its data in the cache with key name **empInfo**, and giving the same data back when we make the same service call again.

**Note**: Cached data will be available until we clear it off manually, spring will not clear the cache automatically after some time.

We can integrate some cache providers like EhCache, Redis..or something, so that we can get better control on the cached data.

Ref.

[*https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-caching.html*](https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-caching.html)

**Clear the Cache**

Generally from our side we will clean or flush the cache after any update or delete operations, for that we use *@CacheEvict* annotation on the required method.

@RestController

**public** **class** CacheController {

@Autowired

**private** CacheDAO dao;

@RequestMapping("/empDeatils")

**@CacheEvict(value = "cacheCustInfo", allEntries=true)**

**public** List<Employee> remove empl(){

**return** dao.getEmployeeDetails();

}

}

**How to disable cache**

If we would like to disable the cache, no need to remove all the annotations 🙂 just add the below line in the *application.properties* file, it takes care everything for you.

spring.cache.type=none

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 We can refer to this link −

<https://docs.oracle.com/cd/E12058_01/doc/doc.1014/e12030/cron_expressions.htm>

The @EnableScheduling annotation is used to enable the scheduler for our application. This annotation should be added into the main Spring Boot application class file.

@SpringBootApplication

@EnableScheduling

publicclassDemoApplication{

publicstaticvoid 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 \* \* ?")

publicvoid cronJobSch()throwsException{

}

**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)

publicvoid 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

publicclassScheduler{

**@Scheduled(fixedRate =1000)**

publicvoid fixedRateSch(){

SimpleDateFormat sdf =newSimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS");

Date now =newDate();

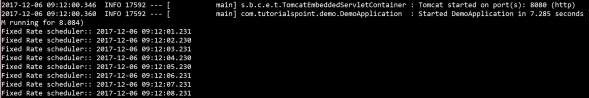
String strDate = sdf.format(now);

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.



**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)

publicvoid 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

publicclassScheduler{

@Scheduled(fixedDelay =1000, initialDelay =3000)

publicvoid fixedDelaySch(){

SimpleDateFormat sdf =newSimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS");

Date now =newDate();

String strDate = sdf.format(now);

System.out.println("Fixed Delay scheduler:: "+ strDate);

}

}

@SpringBootApplication

@EnableScheduling

**public** **class** SpringBootScheduler {

**public** **static** **void** main(String[] args) {

SpringApplication sa = **new** SpringApplication(SpringBootScheduler.**class**);

sa.run(args);

System.***out***.println("Server has been started Successfully!!");

}

}

**package** com.scheduler.component;

**import** java.text.SimpleDateFormat;

**import** java.util.Date;

**import** org.springframework.scheduling.annotation.Scheduled;

**import** org.springframework.stereotype.Component;

@Component("CronJob")

**public** **class** CronJob {

//\*/10 - means every 10 second

@Scheduled(cron = "\* 0/10 \* \* \* ?")

**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("\*\*\*cronJobSch\*\*\*\*\*Java cron job expression:: " + strDate);

}

//this method will being called every minute first time it will called just server has started

@Scheduled(fixedDelay=60000)

**public** **void** fixedDealyed(){

SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS");

Date now = **new** Date();

String strDate = sdf.format(now);

System.***out***.println("Fixed Dealyed :: " + strDate);

}

//this method will being called every value mentioned in the properties file first time it will called just server startup,reading the values from the properties file

@Scheduled(fixedDelayString="${schedule.fixedDelayTime}")

**public** **void** fixedDealyString(){

SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS");

Date now = **new** Date();

String strDate = sdf.format(now);

System.***out***.println("fixedDelayString Dealyed :: " + strDate);

}

/\*

this method will being called every value mentioned in the properties file once the server has started it will wait first time for 30 second.

\*/

@Scheduled(fixedDelayString="${schedule.fixedDelayTime}" , initialDelay=30000)

**public** **void** fixedDealyStringWithInitailDelay(){

SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS");

Date now = **new** Date();

String strDate = sdf.format(now);

System.***out***.println("fixedDelayString with initial Delayed :: " + strDate);

}

@Scheduled(fixedDelayString="${schedule.fixedDelayTime}" , initialDelayString="${schedule.initialwaitTime}")

**public** **void** fixedDealyStringWithInitailDelay11111111(){

SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS");

Date now = **new** Date();

String strDate = sdf.format(now);

System.***out***.println("fixedDelayString with initial Delayed :: " + strDate);

}

}

/\*

Once the server start up then conJobSch() method will automatically called

0/10 = it will call the method in every 10 second

@Scheduled(cron = "1 2 3 4 5 6 7")

Seconds YES 0-59 , - \* /

Minutes YES 0-59 , - \* /

Hours YES 0-23 , - \* /

Day of month YES 1-31 , - \* ? / L W

Month YES 1-12 or JAN-DEC , - \* /

Day of week YES 1-7 or SUN-SAT , - \* ? / L #

Year NO empty, 1970-2099 , - \* /

\* (“all values”)

? (“no specific value”)

- used to specify ranges. For example, “10-12” in the hour field means “the hours 10, 11 and 12”.

\*/

**How to Deploy Spring Boot Applications on External Tomcat Server**

Generally in the real-time projects we won’t use inbuilt servers provided by the frameworks because of many reasons like security, maintenance and control.

Just do these changes to our spring boot application which we want to deploy it on to external tomcat server?

* pom.xml, add dependency and packaging to war
* Extend our main class with SpringBootServletInitializer and override its *configure* method
* Generate WAR and deploy into the external server

We can change the context path also in the application.properties. (Optional)

If you observe the context path in the URL, its showing SpringBootAppInExternalTomcat-0.0.1-SNAPSHOT. Its not taking the context path we have given in the application.properties, rather it’s considering the context path as Artifact Id + Version from pom.xml.

URL: http://localhost:8090/16ExternalTomcatServerConfiguration/external

**Spring Boot - Interceptor**

We 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, we 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, we need to create **@Component** class that supports it and it should implement the **HandlerInterceptor** interface.

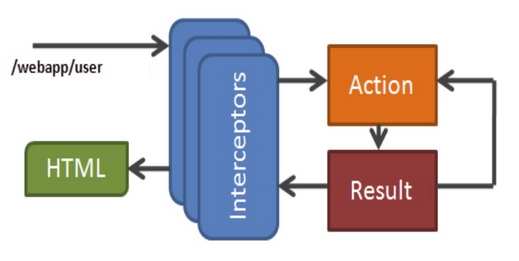
The following are the three methods we should know about while working on Interceptors −

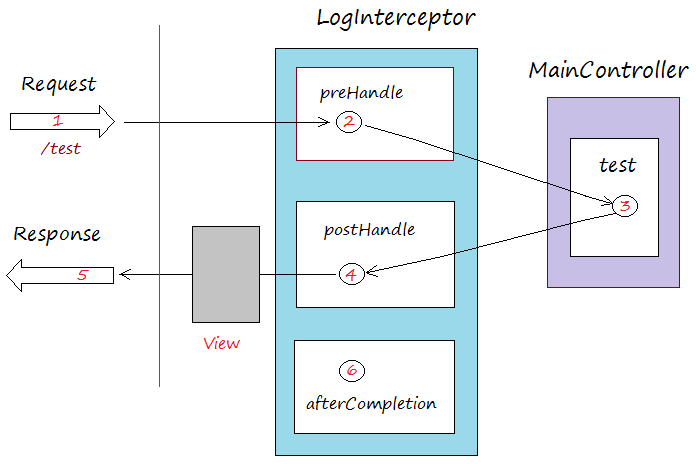
* **preHandle()** method − This is used to perform operations before sending the request to the controller. This method should return true to return the response to the client.
* **postHandle()** method − This is used to perform operations before sending the response to the client.
* **afterCompletion()**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

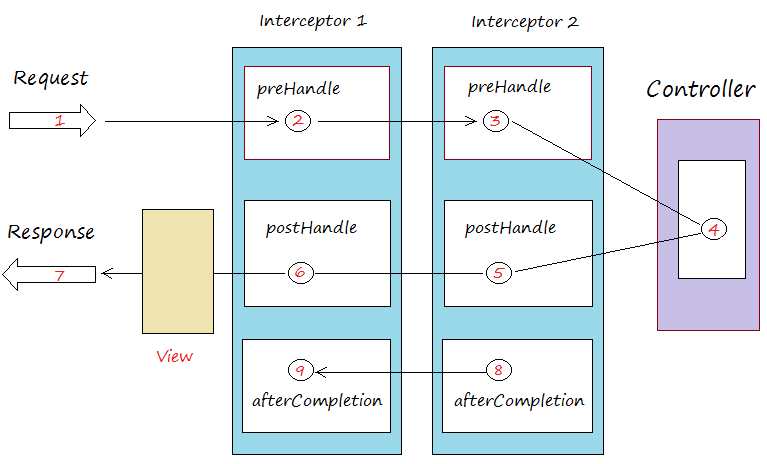
**What is Spring Interceptor?**

When we come to a company and want to meet the company's manager. We need to pass through interceptors which can be a gatekeeper, a receptionist,   
In **Spring**, when a request is sent to spring controller; it will have to pass through **Interceptors** (0 or more) before being processed by **Controller**.

**Spring Interceptor** is a concept that is rather similar to **Servlet Filter.**   
**Spring Interceptor** is only applied to requests that are sending to a **Controller**.







@SpringBootApplication

**public** **class** SpringBootInterceptor {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(SpringBootInterceptor.**class**, args);

System.***out***.println("Server started successfully !!");

}

}

**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;

}

}

@Component

**public** **class** InterceptorConfig **extends** WebMvcConfigurerAdapter {

@Autowired

**public** FirstIntercpetor firstIntercpetor;

@Autowired

**public** SecondInterceptor secondIntercepto;

@Autowired

**public** ThirdInterceptor ThirdInterceptor;

@Override

**public** **void** addInterceptors(InterceptorRegistry registry) {

registry.addInterceptor(firstIntercpetor);

registry.addInterceptor(secondIntercepto);

registry.addInterceptor(ThirdInterceptor);

}

}

/\*

We will have to register this Interceptor with InterceptorRegistry by using WebMvcConfigurerAdapter

as shown in the above

\*/

@RestController

**public** **class** InterceptorController {

**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() {

System.***out***.println("\*\*\*From get product method from Interceptor Controller\*\*\*\*\*\*");

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

}

}

@Component

**public** **class** FirstIntercpetor **implements** HandlerInterceptor {

@Override

**public** **boolean** preHandle(HttpServletRequest req, HttpServletResponse res,

Object handler) **throws** Exception {

System.***out***.println("Pre Handle method is Calling from --FirstInterceptor--- class");

**return** **true**;

}

@Override

**public** **void** postHandle(HttpServletRequest req, HttpServletResponse res,

Object obj, ModelAndView modelAndView) **throws** Exception {

System.***out***.println("Post Handle method is Calling from --FirstInterceptor--- class");

}

@Override

**public** **void** afterCompletion(HttpServletRequest request,

HttpServletResponse response, Object handler, Exception exception)

**throws** Exception {

System.***out***.println("afterCompletion Handle method is Calling from --FirstInterceptor-class");

}

}

@Component

**public** **class** SecondInterceptor **implements** HandlerInterceptor {

@Override

**public** **boolean** preHandle(HttpServletRequest req, HttpServletResponse res,

Object handler) **throws** Exception {

System.***out***.println("Pre Handle method is Calling from --SecondInterceptor--- class");

**return** **true**;

}

@Override

**public** **void** postHandle(HttpServletRequest req, HttpServletResponse res,

Object obj, ModelAndView modelAndView) **throws** Exception {

System.***out***.println("Post Handle method is Calling from -- SecondInterceptor --- class");

}

@Override

**public** **void** afterCompletion(HttpServletRequest request,

HttpServletResponse response, Object handler, Exception exception)

**throws** Exception {

System.***out***.println("afterCompletion Handle method is Calling from -- SecondInterceptor -class");

}

}

@Component

**public** **class** ThirdInterceptor **implements** HandlerInterceptor {

@Override

**public** **boolean** preHandle(HttpServletRequest req, HttpServletResponse res,

Object handler) **throws** Exception {

System.***out***.println("Pre Handle method is Calling from --ThirdInterceptor--- class");

**return** **true**;

}

@Override

**public** **void** postHandle(HttpServletRequest req, HttpServletResponse res,

Object obj, ModelAndView modelAndView) **throws** Exception {

System.***out***.println("Post Handle method is Calling from -- ThirdInterceptor --- class");

}

@Override

**public** **void** afterCompletion(HttpServletRequest request,

HttpServletResponse response, Object handler, Exception exception)

**throws** Exception {

System.***out***.println("afterCompletion Handle method is Calling from -- ThirdInterceptor -class");

}

}

Output

Pre Handle method is calling from --FirstInterceptor--- class

Pre Handle method is calling from -----SecondInterceptor----- class

Pre Handle method is calling from -----ThirdInterceptor----- class

\*\*\*\*\*\*\*\*\*\*\*From get product method from Interceptor Controller \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Post Handle method is calling from -----ThirdInterceptor----- class

Post Handle method is calling from -----SecondInterceptor----- class

Post Handle method is calling from --FirstInterceptor--- class

afterCompletion Handle method is Calling from -----ThirdInterceptor----- class

afterCompletion Handle method is Calling from -----SecondInterceptor----- class

afterCompletion Handle method is Calling from --FirstInterceptor--- class

**Spring Boot - Servlet Filter**

A filter is an object used to intercept the HTTP requests and responses of our 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.

@SpringBootApplication

**public** **class** SpringBootServletFilterApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(SpringBootServletFilterApplication.**class**, args);

System.***out***.println("Spring Boot Servelt Filter Applicatin has started successfully !!!!");

}

}

**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** ServletFilter **implements** Filter {

@Override

**public** **void** init(FilterConfig config) **throws** ServletException {

System.***out***.println("\*\*\*I am from init method of Servelt Filter\*\*\*");

}

@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());

System.***out***.println("\*\*\*I am from doFilter method of Servelt Filter\*\*\*");

filterChain.doFilter(request, response);

}

@Override

**public** **void** destroy() {

System.***out***.println("\*\*\*I am from destroy method of Servelt Filter\*\*\*");

}

}

Application.property file

server.port=2026

server.contextPath=/contextFilter

**package** com.servletFilter.controller;

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

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

@RestController

**public** **class** ServeltController {

@RequestMapping(value = "/filter")

**public** String filterHelloWorld() {

System.***out***.println("I am from filterHelloWorld of ServletController--");

**return** "Filter Hello World";

}

}

Output:

\*\*\*I am from init method of Servelt Filter\*\*\*

Spring Boot Servelt Filter Applicatin has started successfully!!!!

Remote Host:127.0.0.1

Remote Address:127.0.0.1

\*\*\*I am from doFilter method of Servelt Filter\*\*\*

I am from filterHelloWorld of ServletController—

**Spring Boot - Logging**

Spring Boot uses Apache Commons logging for all internal logging.

Spring Boot’s default configurations provide 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 we 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.

2019-04-24 06:34:18.109 INFO 3436 ---[restartedMain] com.log.SpringBootLoggings : this is a info message

2019-04-24 06:34:18.109 WARN 3436 ---[restartedMain] com.log.SpringBootLoggings : this is a warn message

2019-04-24 06:34:18.110 ERROR 3436 ---[restartedMain]com.log.SpringBootLoggings : this is a error message

2019-04-24 06:34:18.110 INFO 3436 ---[restartedMain] com.log.SpringBootLoggings : SpringBootLoggings Started Successfully !!

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 we 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**

We can also add the debug mode to your application.properties file as shown below −

**debug = true**

## File Log Output

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

We can specify the log file path using the property shown below.

Note if we are not specified the log names than the default logger file name is: spring.log.

logging.path = /var/tmp/

We can specify the own log file name using the property shown below −

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

**Note** − 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”.

We 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.

Configure for 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"*>

<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=*"STDOUT"* />

</root>

</configuration>

Configure for File appender in Logback.xml file given below

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

<configuration>

<appender name=*"FILE"* class=*"ch.qos.logback.core.FileAppender"*>

<File>E:/SpringSTSProgram/21Loggings02/manual.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"* />

</root>

</configuration>

Output

[2019-04-24T07:06:44.044Z] [com.log.SpringBootLoggings02] [restartedMain] [16] [INFO ]this is a info message

[2019-04-24T07:06:44.044Z] [com.log.SpringBootLoggings02] [restartedMain] [17] [WARN ]this is a warn message

[2019-04-24T07:06:44.044Z] [com.log.SpringBootLoggings02] [restartedMain] [18] [ERROR]this is a error message

[2019-04-24T07:06:44.044Z] [com.log.SpringBootLoggings02] [restartedMain] [19] [INFO ] SpringBootLoggings Started Successfully !!

**Spring Boot - Actuator**

Spring boot’s module Actuator allows us to monitor and manage application usages in production environment, without coding and configuration for any of them.

These monitoring and management information is exposed via [REST](https://restfulapi.net) like endpoint URLs.

Spring Boot Actuator provides secured endpoints for monitoring and managing our Spring Boot application.

By default, all actuator endpoints are secured. In this chapter, you will learn in detail about how to enable Spring Boot actuator to your application.

## Enabling Spring Boot Actuator

To enable Spring Boot actuator endpoints to your Spring Boot application, we need to add the Spring Boot Starter actuator dependency in our build configuration file.

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

#### Actuator Security with Properties

By default, [spring security](https://howtodoinjava.com/spring-security-tutorial/) is enabled for all actuator endpoints.

It is inbuilt [form-based authentication](https://howtodoinjava.com/spring/spring-boot/role-based-security-jaxrs-annotations/) with the userid as the user and a randomly generated password.

The following entries are then required to customize [basicauth security](https://howtodoinjava.com/spring/spring-security/http-basic-authentication-example-using-spring-3/) to your sensitive endpoints.

application.properties

server.contextPath=/actuator

server.port=2020

management.security.enabled =true

management.security.roles =ADMIN

security.basic.enabled =true

security.user.name =admin

security.user.password =admin

#### Important Actuator Endpoints

|  |  |
| --- | --- |
| **ENDPOINTS** | **USAGE** |
| /beans | To view the Spring beans and its types, scopes and dependency. |
| /info | To view the information about the Spring Boot application. |
| /trace | To view the list of Traces of your Rest endpoints. |
| /env | Returns list of properties in current environment |
| /health | Returns application health information. |
| /auditevents | Returns all auto-configuration candidates and the reason why they ‘were’ or ‘were not’ applied. |
| /trace | Returns trace logs (by default the last 100 HTTP requests). |
| /dump | It performs a thread dump. |
| /metrics | It shows several useful metrics information like JVM memory used, system CPU usage, open files, and much more. |

## Actuator Advance Configuration Options

#### 4.1. Change the Management endpoint context path

By default all endpoints comes in default context path of the application. Still if we need to expose those endpoints in different endpoint, then we need to specify those in application.properties.

|  |
| --- |
| management.context-path=/manage |

Now you will be able to access all actuator endpoints under new URL. e.g.

* /manage/health
* /manage/dump
* /manage/env
* /manage/beans

#### Customize the management server port

To customize the management endpoint port, we need to add this entry in the application.properties file.

|  |
| --- |
| management.port=2030 |

**URL:** [**http://localhost:2030/manage/env**](http://localhost:2030/manage/env)

Spring Boot – Servlet Mapping

Servlet mapping can be achieved either by using ServletRegistrationBean or by using @ServletComponentScan annotation in Spring Boot.

ServletRegistrationBean registers Servlet as spring bean.

@ServletComponentScan scans Servlet annotated with @WebServlet.

The annotation @ServletComponentScan works only using embedded server in Spring Boot.

**Pom.xml**

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.0.3.RELEASE</version>

</parents>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

## Registering Servlets as Spring Beans using ServletRegistrationBean

ServletRegistrationBean is used to register Servlets in Servlet 3.0 + container.

We need to create a bean of ServletRegistrationBean in our JavaConfig.

**setServlet()**: Sets the servlet to be registered.   
**addUrlMappings()**: Add URL mappings for the Servlet.   
**setLoadOnStartup**: Sets priority to load Servlet on startup.

@Configuration

**public** **class** ServetWebConfig {

@Bean

**public** ServletRegistrationBean<HttpServlet> countryServlet() {

ServletRegistrationBean<HttpServlet> bean = **new** ServletRegistrationBean<HttpServlet>();

bean.setServlet(**new** HelloCountryServlet());

bean.addUrlMappings("/country/\*");

bean.setLoadOnStartup(1);

**return** bean;

}

@Bean

**public** ServletRegistrationBean<HttpServlet> stateServlet() {

ServletRegistrationBean<HttpServlet> bean = **new** ServletRegistrationBean<HttpServlet>();

bean.setServlet(**new** HelloStateServlet());

bean.addUrlMappings("/state/\*");

bean.setLoadOnStartup(2);

**return** bean;

}

}

Note: Here HelloCountryServlet and HelloStateServelt class registering and mapping with /country and /state url pattern respectively by using ServletRegistrationBean

Same thing we can achieve by

@WebServlet(urlPatterns = "/country/\*", loadOnStartup = 1) on HelloCountryServlet

@WebServlet(urlPatterns = "/state/\*", loadOnStartup = 2) on HelloStateServelt

## Scanning for Servlets using @ServletComponentScan

@ServletComponentScan in Spring Boot will scan Servlets annotated with @WebServlet.

Filters annotated with @WebFilter and Listeners annotated with @WebListener.

The annotation @ServletComponentScan is used on JavaConfig at class level.

@ServletComponentScan scans Servlets, Filters and Listeners only using an embedded web server.

@WebServlet(urlPatterns = "/country/\*", loadOnStartup = 1)

**public** **class** HelloCountryServlet **extends** HttpServlet {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**public** **void** doPost(HttpServletRequest request, HttpServletResponse response) **throws** IOException {

doGet(request, response);

}

**public** **void** doGet(HttpServletRequest request, HttpServletResponse response) **throws** IOException {

response.setContentType("text/html");

PrintWriter out = response.getWriter();

out.println("<h3>Hello India!</h3>");

}

}

@WebServlet(urlPatterns = "/state/\*", loadOnStartup = 2)

**public** **class** HelloStateServlet **extends** HttpServlet {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**public** **void** doPost(HttpServletRequest request, HttpServletResponse response) **throws** IOException {

doGet(request, response);

}

**public** **void** doGet(HttpServletRequest request, HttpServletResponse response) **throws** IOException {

response.setContentType("text/html");

PrintWriter out = response.getWriter();

out.println("<h3>Hello Bangalore!</h3>");

}

}

@SpringBootApplication

@ServletComponentScan

**public** **class** SpringBootServeltMapping {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(SpringBootServeltMapping.**class**, args);

System.***out***.println("SpringBootServeltMapping has been started successfully!!");

}}

@RestController

**public** **class** HelloWorldController {

@RequestMapping("/world")

**public** String helloMsg() {

String msg = "Hello World!";

**return** msg;

}

}

URL:

<http://localhost:2020/country>

<http://localhost:2020/world>

<http://localhost:2020/state>

Spring Boot Starters

Starters are a set of convenient dependency descriptors which we can include in our application.

Spring Boot provides built-in starters which makes development easier and rapid.

For example, if we want to get started using spring and JPA for database access, just include the **spring-boot-starter-data-jpa** dependency in our project.

Starter should follow a naming pattern like: **spring-boot-starter**-\*, where \* is a particular type of application. This naming structure is intended to help when we need to find a starter.

Spring Boot technical starters

|  |  |
| --- | --- |
| Name | Description |
| spring-boot-starter-undertow | It is used for Undertow as the embedded servlet container. An alternative to spring-boot-starter-tomcat. |
| spring-boot-starter-jetty | It is used for Jetty as the embedded servlet container. An alternative to spring-boot-starter-tomcat. |
| spring-boot-starter-logging | It is used for logging using Logback. Default logging starter. |
| spring-boot-starter-tomcat | It is used for Tomcat as the embedded servlet container. Default servlet container starter used by spring-boot-starter-web. |
| spring-boot-starter-log4j2 | It is used for Log4j2 for logging. An alternative to spring-boot-starter-logging. |

### ****Query Parameter vs URI Path - VVI****

@RequestParams extract values from the query string

@GetMapping(value="/empDetailsUsingRequestParam")

@ResponseBody

**public** Employee getEmpById2(@RequestParam String id){

**return** dao.getEmpById(**new** BigDecimal(id));

}

//http://localhost:2085/dataSource/empDetailsUsingRequestParam?id=123 = QUERY STRING

@PathVariables extract values from the URI path:

@GetMapping(value="/empDetailsUsingPathVariable/{empId}")

@ResponseBody

**public** Employee getEmpById(@PathVariable String empId){

**return** dao.getEmpById(**new** BigDecimal(empId));

}

//http://localhost:2085/dataSource/empDetailsUsingPathVariable/123 = URI PATH

| **HTTP Method** | **URI** | **Description** |
| --- | --- | --- |
| **GET** | **/books** | **List all books.** |
| **POST** | **/books** | **Save a book.** |
| **GET** | **/books/{id}** | **Find a book where id = {:id}.** |
| **PUT** | **/books/{id}** | **Update a book where id = {:id}, or save it.** |
| **PATCH** | **/books/{id}** | **Update a single field where id = {:id}.** |
| **DELETE** | **/books/{id}** | **Delete a book where id = {:id}.** |

### ****@SpringBootApplication Annotation parameters****

**@SpringBootApplication** annotation takes up four optional parameters

* **exclude:** This parameter excludes the list of the classes from the auto-configuration
* **excludeNames:** This parameter excludes the list of fully qualified class names from the auto configuration
* **scanBasePackageClasses:** This parameters provides the list of classes in the other packages to which has to be applied for scanning
* **scanBasePackages:** This parameters provides the list of packages which has to be applied for scanning

[**@SpringBootApplication**](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/autoconfigure/SpringBootApplication.html) annotation [from Spring Boot 1.2.0] in order to enable **auto-configuration** feature, the **@SpringBootApplication** annotation performs the work of the three annotations **@Configuration, @ComponentScan** and **@EnableAutoConfiguration.**

* **@Configuration –** Allows you to register beans or import additional configuration class
* **@ComponentScan –** Enables Component scanning of the packages specified
* **@EnableAutoConfiguration –** Enables Spring Boot’s auto-configuration feature

In short **@SpringBootApplication** annotation in equivalent to use @Configuration, @ComponentScan, @EnableAutoConfiguration annotations with their default configuration.

**Difference between POST, PUT and PATCH methods of a HTTP protocol**

**POST**

A HTTP.POST method always creates a new resource on the server. Its a non-idempotent request i.e. if user hits same requests 2 times it would create another new resource if there is no constraint.

http post method is like a INSERT query in SQL which always creates a new record in database.

Example: Use POST method to save new user, order etc where backend server decides the resource id for new resource.

**PUT**

In HTTP.PUT method the resource is first identified from the URL and if it exists then it is updated otherwise a new resource is created. When the target resource exists it overwrites that resource with a complete new body. That is HTTP.PUT method is used to CREATE or UPDATE a resource.

http put method is like a MERGE query in SQL which inserts or updates a record depending upon whether the given record exists.

PUT request is idempotent i.e. hitting the same requests twice would update the existing recording (No new record created). In PUT method the resource id is decided by the client and provided in the request url.

Example: Use PUT method to update existing user or order.

**PATCH**

A HTTP.PATCH method is used for partial modifications to a resource i.e. delta updates.

http patch method is like a UPDATE query in SQL which sets or updates selected columns only and not the whole row.

**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.

**Spring Boot - Exception Handling**

Handling exceptions and errors in APIs and sending the proper response to the client is good for enterprise applications.

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.

We can use the following code to create @ControllerAdvice class to handle the exceptions globally –

### Response Statuses for Errors

Use appropriate status code based on the error.

* 404 - RESOURCE NOT FOUND
* 400 - BAD REQUEST
* 401 - UNAUTHORIZED
* 415 - UNSUPPORTED TYPE - Representation not supported for the resource
* 500 - SERVER ERROR

Let’s consider a few HTTP Methods:

* GET : Should not update anything. Should be idempotent (same result in multiple calls). Possible Return Codes 200 (OK) + 404 (NOT FOUND) +400 (BAD REQUEST)
* POST : Should create new resource. Ideally return JSON with link to newly created resource. Same return codes as get possible. In addition - Return code 201 (CREATED) can be used.
* PUT : Update a known resource. ex: update client details. Possible Return Codes : 200(OK) + 404 (NOT FOUND) +400 (BAD REQUEST)
* DELETE: Used to delete a resource. Possible Return Codes : 200(OK).

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

@ControllerAdvice

**public** **class** ProductExceptionController {

}

@Async will make it execute in a separate thread, i.e. the caller will not wait for the completion of the called method.

<https://dzone.com/articles/spring-boot-creating-asynchronous-methods-using-as>