**Spring Boot Tutorial**

* Spring Boot Tutorial provides basic and advanced concepts of Spring Framework. Our Spring Boot Tutorial is designed for beginners and professionals both.
* Spring Boot is a Spring module that provides the RAD (Rapid Application Development) feature to the Spring framework.
* Our Spring Boot Tutorial includes all topics of Spring Boot such, as features, project, maven project, starter project wizard, Spring Initializr, CLI, applications, annotations, dependency management, properties, starters, Actuator, JPA, JDBC, etc.

**What is Spring Boot**

* Spring Boot is a project that is built on the top of the Spring Framework.
* It provides an easier and faster way to set up, configure, and run both simple and web-based applications.
* It is a Spring module that provides the **RAD (*Rapid Application Development*)** feature to the Spring Framework. It is used to create a stand-alone Spring-based application that you can just run because it needs minimal Spring configuration.



* In short, Spring Boot is the combination of **Spring Framework** and **Embedded Servers**
* In Spring Boot, there is no requirement for XML configuration (deployment descriptor).
* It uses convention over configuration software design paradigm that means it decreases the effort of the developer
* We can use Spring **STS IDE** or **Spring Initializr** to develop Spring Boot Java applications.

**Why should we use Spring Boot Framework?**

We should use Spring Boot Framework because:

* The dependency injection approach is used in Spring Boot.
* It contains powerful database transaction management capabilities.
* It simplifies integration with other Java frameworks like JPA/Hibernate ORM, Struts, etc.
* It reduces the cost and development time of the application.

Along with the Spring Boot Framework, many other Spring sister projects help to build applications addressing modern business needs. There are the following Spring sister projects are as follows:

* **Spring Data:** It simplifies data access from the relational and **NoSQL** databases.
* **Spring Batch:** It provides powerful **batch** processing.
* **Spring Security:** It is a security framework that provides robust **security** to applications.
* **Spring Social:** It supports integration with **social networking** like LinkedIn.
* **Spring Integration:** It is an implementation of Enterprise Integration Patterns. It facilitates integration with other **enterprise applications** using lightweight messaging and declarative adapters.

Advantages of Spring Boot

* It creates **stand-alone** Spring applications that can be started using Java **-jar**

.

* It tests web applications easily with the help of different **Embedded** HTTP servers such as **Tomcat, Jetty,** etc. We don't need to deploy WAR files.
* It provides opinionated '**starter**' POMs to simplify our Maven configuration.
* It provides **production-ready** features such as **metrics, health checks,** and **externalized configuration**.
* There is no requirement for **XML** configuration.
* It offers a **CLI** tool for developing and testing the Spring Boot application.
* It offers the number of **plug-ins**.
* It also minimizes writing multiple **boilerplate codes** (the code that has to be included in many places with little or no alteration), XML configuration, and annotations.
* It **increases productivity** and reduces development time.

Limitations of Spring Boot

* Spring Boot can use dependencies that are not going to be used in the application.
* These dependencies increase the size of the application.

Goals of Spring Boot

The main goal of Spring Boot is to reduce **development, unit test,** and **integration test** time.

* Provides Opinionated Development approach
* Avoids defining more Annotation Configuration
* Avoids writing lots of import statements
* Avoids XML Configuration.

By providing or avoiding the above points, Spring Boot Framework reduces **Development time, Developer Effort,** and **increases productivity**.

Prerequisite of Spring Boot

To create a Spring Boot application, following are the prerequisites. In this tutorial, we will use **Spring Tool Suite** (STS) IDE.

* Java 1.8
* Maven 3.0+
* Spring Framework 5.0.0.BUILD-SNAPSHOT
* An IDE (Spring Tool Suite) is recommended.

Spring Boot Features

* Web Development
* SpringApplication
* Application events and listeners
* Admin features
* Externalized Configuration
* Properties Files
* YAML Support
* Type-safe Configuration
* Logging
* Security

**Web Development**

* It is a well-suited Spring module for web application development.
* We can easily create a self-contained HTTP application that uses embedded servers like **Tomcat, Jetty,** or Undertow.
* We can use the **spring-boot-starter-web** module to start and run the application quickly.

**SpringApplication**

* The SpringApplication is a class that provides a convenient way to bootstrap a Spring application.
* It can be started from the main method. We can call the application just by calling a static run() method.

1. **public** **static** **void** main(String[] args)
2. {
3. SpringApplication.run(ClassName.**class**, args);
4. }

**Application Events and Listeners**

* Spring Boot uses events to handle the variety of tasks.
* It allows us to create factories file that is used to add listeners.
* We can refer it to using the **ApplicationListener key**.

Always create factories file in META-INF folder like **META-INF/spring.factories**.

**Admin Support**

* Spring Boot provides the facility to enable admin-related features for the application.
* It is used to access and manage applications remotely.
* We can enable it in the Spring Boot application by using **spring.application.admin.enabled** property.

**Externalized Configuration**

* Spring Boot allows us to externalize our configuration so that we can work with the same application in different environments.
* The application uses YAML files to externalize configuration.

**Properties Files**

* Spring Boot provides a rich set of **Application Properties**.
* So, we can use that in the properties file of our project.
* The properties file is used to set properties like **server-port =8080 or 8081** and many others. It helps to organize application properties.

**YAML Support**

* It provides a convenient way of specifying the hierarchical configuration.
* It is a superset of JSON. The SpringApplication class automatically supports YAML.
* It is an alternative of properties file.

**Type-safe Configuration**

* The strong type-safe configuration is provided to govern and validate the configuration of the application.
* Application configuration is always a crucial task which should be type-safe.
* We can also use annotation provided by this library.

**Logging**

* Spring Boot uses Common logging for all internal logging.
* Logging dependencies are managed by default.
* We should not change logging dependencies if no customization is needed.

**Security**

* Spring Boot applications are spring bases web applications.
* So, it is secure by default with basic authentication on all HTTP endpoints.
* A rich set of Endpoints is available to develop a secure Spring Boot application.

|  |  |
| --- | --- |
| **Spring** | **Spring Boot** |
| **Spring Framework** is a widely used Java EE framework for building applications  . | **Spring Boot Framework** is widely used to develop **REST APIs**. |
| It aims to simplify Java EE development that makes developers more productive. | It aims to shorten the code length and provide the easiest way to develop **Web Applications**. |
| The primary feature of the Spring Framework is **dependency injection**. | The primary feature of Spring Boot is **Autoconfiguration**. It automatically configures the classes based on the requirement. |
| It helps to make things simpler by allowing us to develop **loosely coupled** applications. | It helps to create a **stand-alone** application with less configuration. |
| The developer writes a lot of code (**boilerplate code**) to do the minimal task. | It **reduces** boilerplate code. |
| To test the Spring project, we need to set up the sever explicitly. | Spring Boot offers **embedded server** such as **Jetty** and **Tomcat**, etc. |
| It does not provide support for an in-memory database  . | It offers several plugins for working with an embedded and **in-memory** database such as **H2**. |
| Developers manually define dependencies for the Spring project in **pom.xml**. | Spring Boot comes with the concept of **starter** in pom.xml file that internally takes care of downloading the dependencies **JARs** based on Spring Boot Requirement. |

|  |  |
| --- | --- |
| **Spring Boot** | **Spring MVC** |
| **Spring Boot** is a module of Spring for packaging the Spring-based application with sensible defaults  . | **Spring MVC** is a model view controller-based web framework under the Spring framework  . |
| It provides default configurations to build **Spring-powered** framework  . | It provides **ready to use** features for building a web application. |
| There is no need to build configuration manually. | It requires build configuration manually. |
| There is **no requirement** for a deployment descriptor. | A Deployment descriptor is **required**. |
| It avoids boilerplate code and wraps dependencies together in a single unit. | It specifies each dependency separately. |
| It **reduces** development time and increases productivity. | It takes **more** time to achieve the same. |

Spring Boot Architecture

* Spring Boot is a module of the Spring Framework.
* It is used to create stand-alone, production-grade Spring Based Applications with minimum efforts.
* It is developed on top of the core Spring Framework.

Spring Boot follows a layered architecture in which each layer communicates with the layer directly below or above (hierarchical structure) it.

Before understanding the **Spring Boot Architecture**, we must know the different layers and classes present in it. There are **four** layers in Spring Boot are as follows:

* **Presentation Layer**
* **Business Layer**
* **Persistence Layer**
* **Database Layer**



**Presentation Layer:**

* The presentation layer handles the HTTP requests, translates the JSON parameter to object, and authenticates the request and transfer it to the business layer. In short, it consists of **views** i.e., frontend part.

**Business Layer:**

* The business layer handles all the **business logic**.
* It consists of service classes and uses services provided by data access layers.
* It also performs **authorization** and **validation**.

**Persistence Layer:**

* The persistence layer contains all the **storage logic** and translates business objects from and to database rows.

**Database Layer:**

* In the database layer, **CRUD** (create, retrieve, update, delete) operations are performed.

Spring Boot Flow Architecture



* Now we have validator classes, view classes, and utility classes.
* Spring Boot uses all the modules of Spring-like Spring MVC, Spring Data, etc. The architecture of Spring Boot is the same as the architecture of Spring MVC, except one thing: there is no need for **DAO** and **DAOImpl** classes in Spring boot.
* Creates a data access layer and performs CRUD operation.
* The client makes the HTTP requests (PUT or GET).
* The request goes to the controller, and the controller maps that request and handles it. After that, it calls the service logic if required.
* In the service layer, all the business logic performs. It performs the logic on the data that is mapped to JPA with model classes.
* A JSP page is returned to the user if no error occurred.

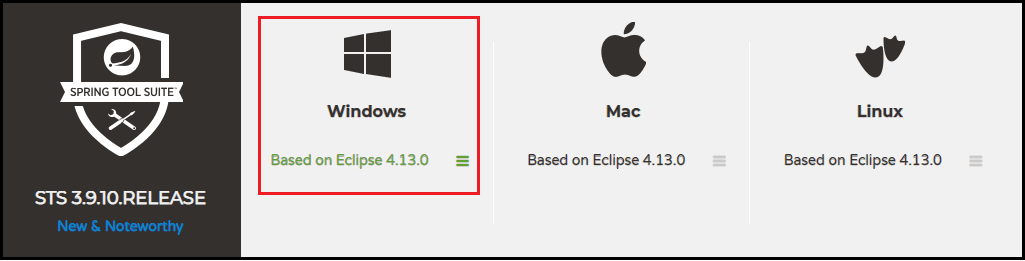
# **Download and Install STS IDE**

## Spring Tool Suite (STS) IDE

* Spring Tool Suite is an IDE to develop Spring applications. It is an Eclipse-based development environment.
* It provides a ready-to-use environment to implement, run, deploy, and debug the application.
* It validates our application and provides quick fixes for the applications.

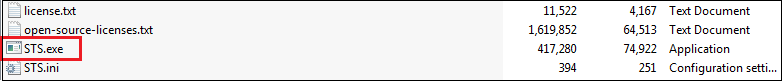
### **Installing STS**

**Step 1:** Download Spring Tool Suite from <https://spring.io/tools3/sts/all>. Click on the platform which you are using. In this tutorial, we are using the Windows platform.

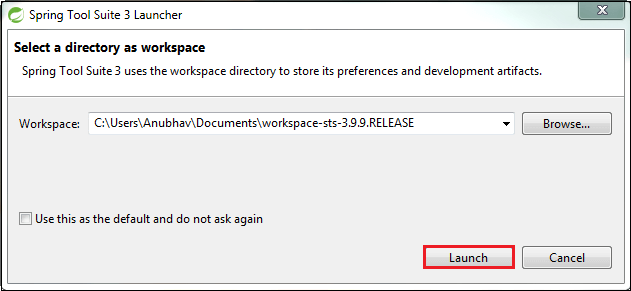


**Step 2:** Extract the **zip** file and install the STS.

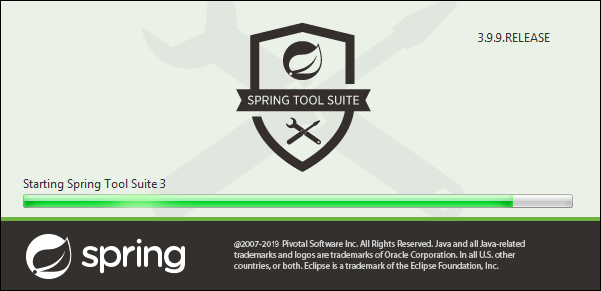
sts-bundle -> sts-3.9.9.RELEASE -> Double-click on the **STS.exe**.



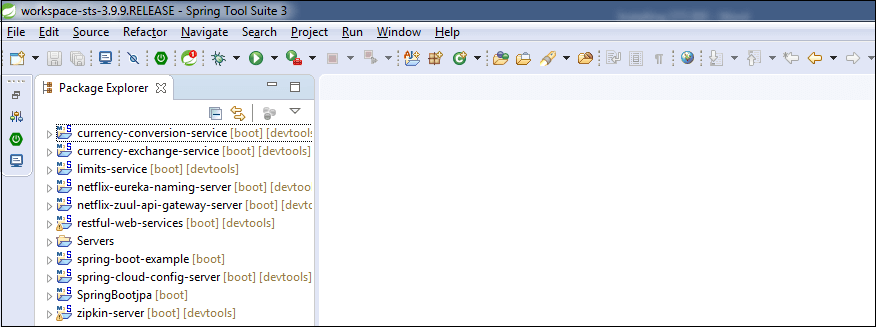
**Step 3:** Spring Tool Suite 3 Launcher dialog box appears on the screen. Click on the **Launch** button. You can change the Workspace if you want.



**Step 4:** It starts launching the STS.



The STS user interface looks like the following:



Creating a Spring Boot Project

Following are the steps to create a simple Spring Boot Project.

**Step 1:** Open the Spring initializr [https://start.spring.io](https://start.spring.io/).

**Step 2:** Provide the **Group** and **Artifact** name. We have provided Group name **com.javatpoint** and Artifact **spring-boot-example**.

**Step 3:** Now click on the **Generate** button.

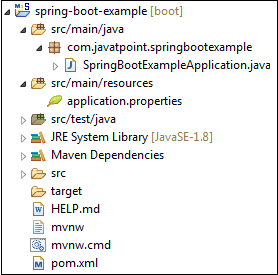
When we click on the Generate button, it starts packing the project in a **.rar** file and downloads the project.

**Step 4:** Extract the **RAR** file.

**Step 5:** **Import** the folder.

File -> Import -> Existing Maven Project -> Next -> Browse -> Select the project -> Finish

It takes some time to import the project. When the project imports successfully, we can see the project directory in the **Package Explorer**. The following image shows the project directory:



**SpringBootExampleApplication.java**

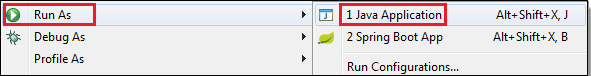
1. **package** com.javatpoint.springbootexample;
2. **import** org.springframework.boot.SpringApplication;
3. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
4. @SpringBootApplication
5. **public** **class** SpringBootExampleApplication
6. {
7. **public** **static** **void** main(String[] args)
8. {
9. SpringApplication.run(SpringBootExampleApplication.**class**, args);
10. }
11. }

**pom.xml**

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<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 https://maven.apache.org/xsd/maven-4.0.0.xsd"**>**
3. **<modelVersion>**4.0.0**</modelVersion>**
4. **<parent>**
5. **<groupId>**org.springframework.boot**</groupId>**
6. **<artifactId>**spring-boot-starter-parent**</artifactId>**
7. **<version>**2.2.2.BUILD-SNAPSHOT**</version>**
8. **<relativePath/>** <!-- lookup parent from repository -->
9. **</parent>**
10. **<groupId>**com.javatpoint**</groupId>**
11. **<artifactId>**spring-boot-example**</artifactId>**
12. **<version>**0.0.1-SNAPSHOT**</version>**
13. **<name>**spring-boot-example**</name>**
14. **<description>**Demo project for Spring Boot**</description>**
15. **<properties>**
16. **<java.version>**1.8**</java.version>**
17. **</properties>**
18. **<dependencies>**
19. **<dependency>**
20. **<groupId>**org.springframework.boot**</groupId>**
21. **<artifactId>**spring-boot-starter**</artifactId>**
22. **</dependency>**
23. **<dependency>**
24. **<groupId>**org.springframework.boot**</groupId>**
25. **<artifactId>**spring-boot-starter-test**</artifactId>**
26. **<scope>**test**</scope>**
27. **<exclusions>**
28. **<exclusion>**
29. **<groupId>**org.junit.vintage**</groupId>**
30. **<artifactId>**junit-vintage-engine**</artifactId>**
31. **</exclusion>**
32. **</exclusions>**
33. **</dependency>**
34. **</dependencies>**
35. **<build>**
36. **<plugins>**
37. **<plugin>**
38. **<groupId>**org.springframework.boot**</groupId>**
39. **<artifactId>**spring-boot-maven-plugin**</artifactId>**
40. **</plugin>**
41. **</plugins>**
42. **</build>**
43. **<repositories>**
44. **<repository>**
45. **<id>**spring-milestones**</id>**
46. **<name>**Spring Milestones**</name>**
47. **<url>**https://repo.spring.io/milestone**</url>**
48. **</repository>**
49. **<repository>**
50. **<id>**spring-snapshots**</id>**
51. **<name>**Spring Snapshots**</name>**
52. **<url>**https://repo.spring.io/snapshot**</url>**
53. **<snapshots>**
54. **<enabled>**true**</enabled>**
55. **</snapshots>**
56. **</repository>**
57. **</repositories>**
58. **<pluginRepositories>**
59. **<pluginRepository>**
60. **<id>**spring-milestones**</id>**
61. **<name>**Spring Milestones**</name>**
62. **<url>**https://repo.spring.io/milestone**</url>**
63. **</pluginRepository>**
64. **<pluginRepository>**
65. **<id>**spring-snapshots**</id>**
66. **<name>**Spring Snapshots**</name>**
67. **<url>**https://repo.spring.io/snapshot**</url>**
68. **<snapshots>**
69. **<enabled>**true**</enabled>**
70. **</snapshots>**
71. **</pluginRepository>**
72. **</pluginRepositories>**
73. **</project>**

**Step 6:** Run the **SpringBootExampleApplication.java** file.

Right-click on the file -> Run As -> Java Applications



The following image shows the application runs successfully.

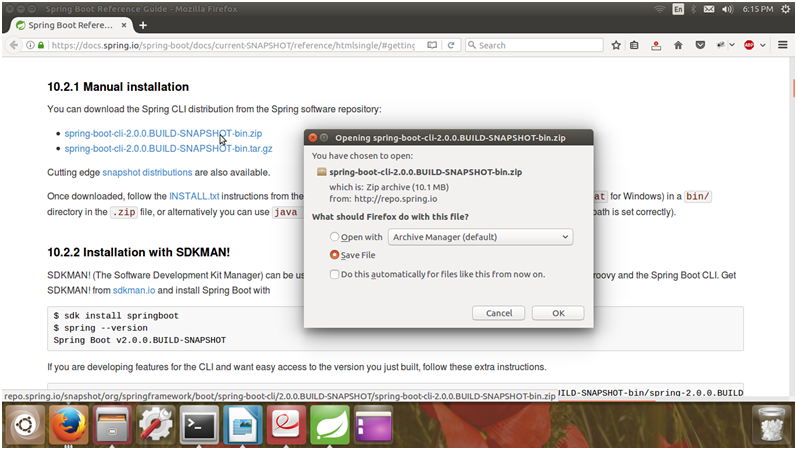
Creating Spring Boot Project

[Download Project](https://static.javatpoint.com/springboot/download/spring-boot-example.zip)

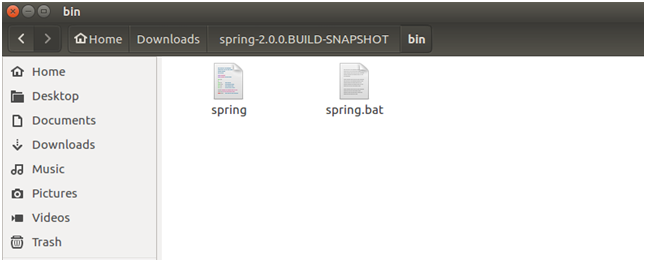
Spring Boot CLI

It is a tool which you can download from the official site of Spring Framework. Here, we are explaining steps.

Download the CLI tool from official site as we are doing here.



After downloading, extract the zip file. It contains a bin folder, in which spring setup is stored. We can use it to execute Spring Boot application.



CLI executes groovy files. So, first, we need to create a groovy file for Spring Boot application.

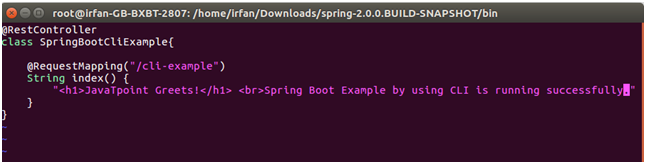
**Open terminal and cd into the bin location of cli folder.**



Create a groovy file.



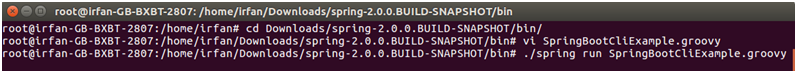
Create a controller in the groovy file.



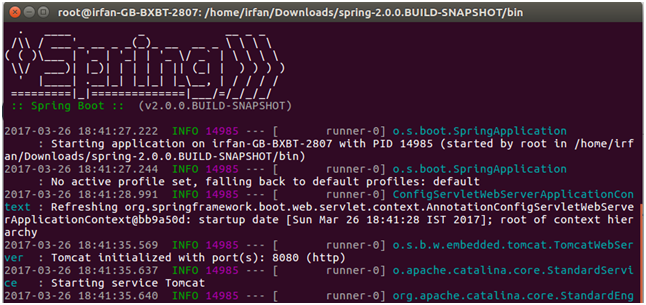
**Execute this file**

By using the following command.

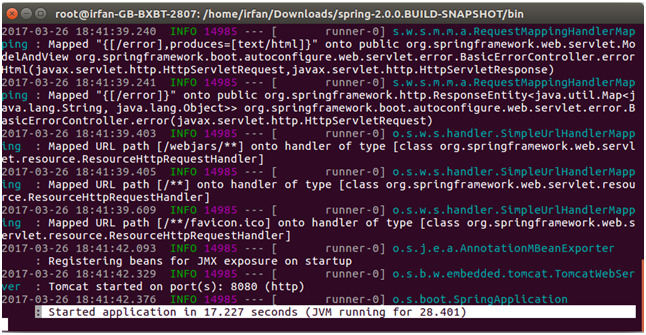
1. ./spring run SpringBootCliExample.groovy



After executing the above command, it starts the execution and produces the following output.



And after lots of lines. It shows the current status of application as follow.



This project is running on the port 8080. So, we can invoke it on any browser by using the following url.

1. localhost:8080:/cli-example

It will produce the following output.

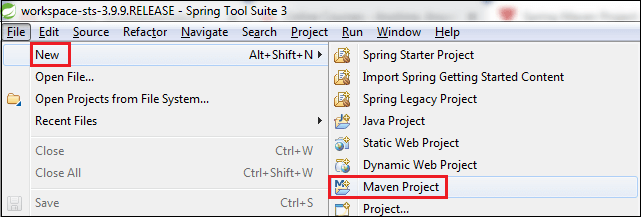


Creating a Spring Boot Project Using STS

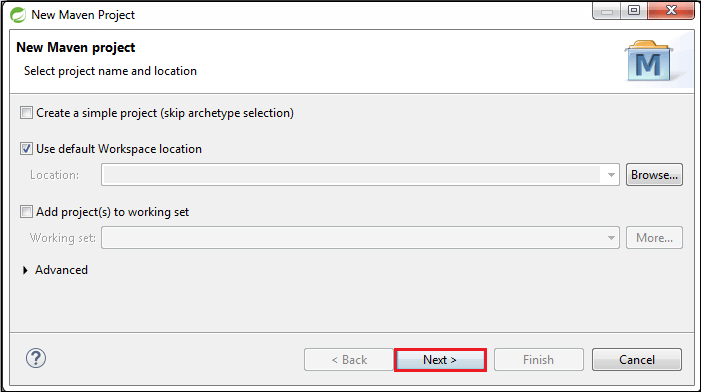
We can also use Spring Tool Suite to create a Spring project. In this section, we will create a **Maven Project** using **STS**.

**Step 1:** Open the Spring Tool Suite.

**Step 2:** Click on the File menu -> New -> Maven Project

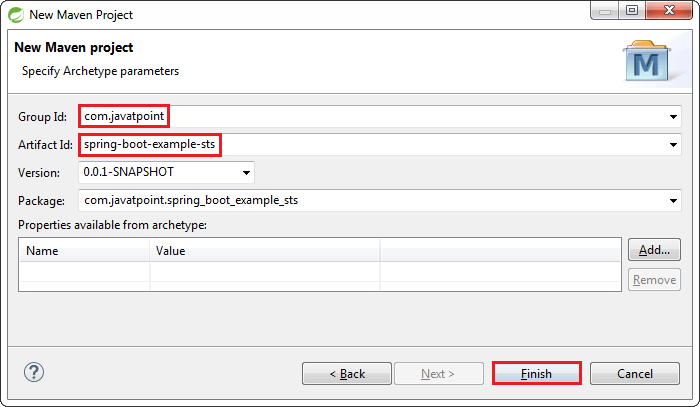


It shows the New Maven Project wizard. Click on the **Next** button.

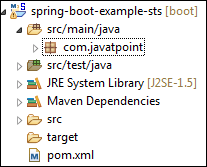


**Step 3:** Select the **maven-archetype-quickstart** and click on the **Next** button.

**Step 4:** Provide the **Group Id** and **Artifact Id**. We have provided Group Id **com.javatpoint** and Artifact Id **spring-boot-example-sts**. Now click on the **Finish** button.



When we click on the Finish button, it creates the project directory, as shown in the following image.



**Step 5:** Open the **App.java** file. We found the following code that is by default.

**App.java**

1. **package** com.javatpoint;
2. **public** **class** App
3. {
4. **public** **static** **void** main( String[] args )
5. {
6. System.out.println( "Hello World!" );
7. }
8. }

The Maven project has a **pom.xml** file which contains the following default configuration.

**pom.xml**

1. **<project** xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>**
3. **<modelVersion>**4.0.0**</modelVersion>**
4. **<groupId>**com.javatpoint**</groupId>**
5. **<artifactId>**spring-boot-example-sts**</artifactId>**
6. **<version>**0.0.1-SNAPSHOT**</version>**
7. **<packaging>**jar**</packaging>**
8. **<name>**spring-boot-example-sts**</name>**
9. **<url>**http://maven.apache.org**</url>**
10. **<properties>**
11. **<project.build.sourceEncoding>**UTF-8**</project.build.sourceEncoding>**
12. **</properties>**
13. **<dependencies>**
14. **<dependency>**
15. **<groupId>**junit**</groupId>**
16. **<artifactId>**junit**</artifactId>**
17. **<version>**3.8.1**</version>**
18. **<scope>**test**</scope>**
19. **</dependency>**
20. **</dependencies>**
21. **</project>**

**Step 6:** Add **Java version** inside the **<properties>** tag.

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

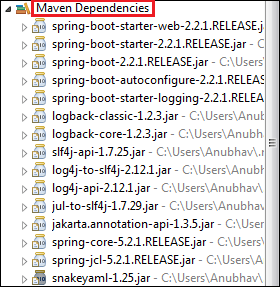
**Step 7:** In order to make a Spring Boot Project, we need to configure it. So, we are adding **spring boot starter parent** dependency in **pom.xml** file. Parent is used to declare that our project is a child to this parent project.

1. **<dependency>**
2. **<groupId>**org.springframework.boot**</groupId>**
3. **<artifactId>**spring-boot-starter-parent**</artifactId>**
4. **<version>**2.2.1.RELEASE**</version>**
5. **<type>**pom**</type>**
6. **</dependency>**

**Step 8:** Add the **spring-boot-starter-web** dependency in **pom.xml** file.

1. **<dependency>**
2. **<groupId>**org.springframework.boot**</groupId>**
3. **<artifactId>**spring-boot-starter-web**</artifactId>**
4. **<version>**2.2.1.RELEASE**</version>**
5. **</dependency>**

Note: When we add the dependencies in the pom file, it downloads the related jar file. We can see the downloaded jar files in the Maven Dependencies folder of the project directory.



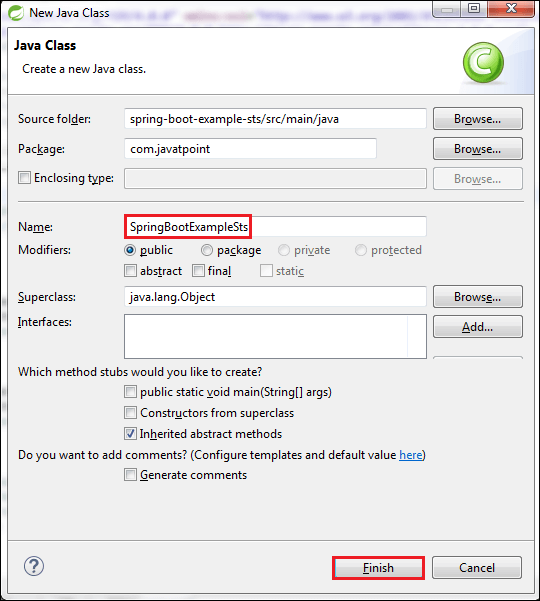
After adding all the dependencies, the pom.xml file looks like the following:

**pom.xml**

1. **<project** xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>**
3. **<modelVersion>**4.0.0**</modelVersion>**
4. **<groupId>**com.javatpoint**</groupId>**
5. **<artifactId>**spring-boot-example-sts**</artifactId>**
6. **<version>**0.0.1-SNAPSHOT**</version>**
7. **<packaging>**jar**</packaging>**
8. **<name>**spring-boot-example-sts**</name>**
9. **<url>**http://maven.apache.org**</url>**
10. **<properties>**
11. **<project.build.sourceEncoding>**UTF-8**</project.build.sourceEncoding>**
12. **<java.version>**1.8**</java.version>**
13. **</properties>**
14. **<dependencies>**
15. **<dependency>**
16. **<groupId>**org.springframework.boot**</groupId>**
17. **<artifactId>**spring-boot-starter-parent**</artifactId>**
18. **<version>**2.2.1.RELEASE**</version>**
19. **<type>**pom**</type>**
20. **</dependency>**
21. **<dependency>**
22. **<groupId>**org.springframework.boot**</groupId>**
23. **<artifactId>**spring-boot-starter-web**</artifactId>**
24. **<version>**2.2.1.RELEASE**</version>**
25. **</dependency>**
26. **<dependency>**
27. **<groupId>**junit**</groupId>**
28. **<artifactId>**junit**</artifactId>**
29. **<version>**3.8.1**</version>**
30. **<scope>**test**</scope>**
31. **</dependency>**
32. **</dependencies>**
33. **</project>**

**Step 9:** Create a class with the name **SpringBootExampleSts** in the package **com.javatpoint**.

Right-click on the package name -> New -> Class -> provide the class name -> Finish



**Step 10:** After creating the class file, call the static method **run()** of the SpringApplication class. In the following code, we are calling the run() method and passing the class name as an argument.

1. SpringApplication.run(SpringBootExampleSts.**class**, args);

**Step 11:** Annotate the class by adding an annotation **@SpringBootApplication**.

**@SpringBootApplication**

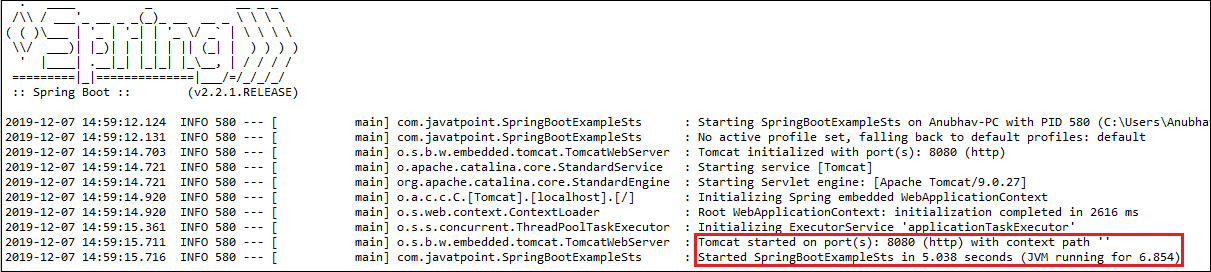
A single @SpringBootApplication annotation is used to enable the following annotations:

* **@EnableAutoConfiguration:** It enables the Spring Boot auto-configuration mechanism.
* **@ComponentScan:** It scans the package where the application is located.
* **@Configuration:** It allows us to register extra beans in the context or import additional configuration classes.

**SpringBootApplicationSts.java**

1. **package** com.javatpoint;
2. **import** org.springframework.boot.SpringApplication;
3. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
4. @SpringBootApplication
5. **public** **class** SpringBootExampleSts
6. {
7. **public** **static** **void** main(String[] args)
8. {
9. SpringApplication.run(SpringBootExampleSts.**class**, args);
10. }
11. }

**Step:** Run the file **SpringBootExampleSts.java**, as Java Application. It displays the following in the console.



The line **Started SpringBootExampleSts in 5.038 seconds (JVM running for 6.854)** in the console shows that the application is up and running.

**Spring Boot Annotations**

* Spring Boot Annotations is a form of metadata that provides data about a program.
* In other words, annotations are used to provide **supplemental** information about a program.
* It is not a part of the application that we develop.
* It does not have a direct effect on the operation of the code they annotate.
* It does not change the action of the compiled program.

In this section, we are going to discuss some important **Spring Boot Annotation** that we will use later in this tutorial.

Core Spring Framework Annotations

**@Required:**

* It applies to the **bean** setter method.
* It indicates that the annotated bean must be populated at configuration time with the required property, else it throws an exception **BeanInitilizationException**.

**Example**

1. **public** **class** Machine
2. {
3. **private** Integer cost;
4. @Required
5. **public** **void** setCost(Integer cost)
6. {
7. **this**.cost = cost;
8. }
9. **public** Integer getCost()
10. {
11. **return** cost;
12. }
13. }

**@Autowired:**

* Spring provides annotation-based auto-wiring by providing @Autowired annotation.
* It is used to autowire spring bean on setter methods, instance variable, and constructor.
* When we use @Autowired annotation, the spring container auto-wires the bean by matching data-type.

**Example**

1. @Component
2. **public** **class** Customer
3. {
4. **private** Person person;
5. @Autowired
6. **public** Customer(Person person)
7. {
8. **this**.person=person;
9. }
10. }

**@Configuration:**

* It is a class-level annotation. The class annotated with @Configuration used by Spring Containers as a source of bean definitions.

**Example**

1. @Configuration
2. **public** **class** Vehicle
3. {
4. @BeanVehicle engine()
5. {
6. **return** **new** Vehicle();
7. }
8. }

**@ComponentScan:**

* It is used when we want to scan a package for beans.
* It is used with the annotation @Configuration.
* We can also specify the base packages to scan for Spring Components

.

**Example**

1. @ComponentScan(basePackages = "com.javatpoint")
2. @Configuration
3. **public** **class** ScanComponent
4. {
5. // ...
6. }

**@Bean:**

* It is a method-level annotation. It is an alternative of XML <bean> tag.
* It tells the method to produce a bean to be managed by Spring Container

.

**Example**

1. @Bean
2. **public** BeanExample beanExample()
3. {
4. **return** **new** BeanExample ();
5. }

Spring Framework Stereotype Annotations

**@Component:**

* It is a class-level annotation.
* It is used to mark a Java class as a bean.
* A Java class annotated with **@Component** is found during the classpath.
* The Spring Framework pick it up and configure it in the application context as a **Spring Bean**

.

**Example**

1. @Component
2. **public** **class** Student
3. {
4. .......
5. }

**@Controller:**

* The @Controller is a class-level annotation.
* It is a specialization of **@Component**.
* It marks a class as a web request handler.
* It is often used to serve web pages.
* By default, it returns a string that indicates which route to redirect. It is mostly used with **@RequestMapping** annotation

.

**Example**

1. @Controller
2. @RequestMapping("books")
3. **public** **class** BooksController
4. {
5. @RequestMapping(value = "/{name}", method = RequestMethod.GET)
6. **public** Employee getBooksByName()
7. {
8. **return** booksTemplate;
9. }
10. }

**@Service:**

* It is also used at class level.
* It tells the Spring that class contains the **business logic**.

**Example**

1. **package** com.javatpoint;
2. @Service
3. **public** **class** TestService
4. {
5. **public** **void** service1()
6. {
7. //business code
8. }
9. }

**@Repository:**

* It is a class-level annotation.
* The repository is a **DAOs** (Data Access Object) that access the database directly.
* The repository does all the operations related to the database.

1. **package** com.javatpoint;
2. @Repository
3. **public** **class** TestRepository
4. {
5. **public** **void** delete()
6. {
7. //persistence code
8. }
9. }

**Spring Boot Annotations**

**@EnableAutoConfiguration:**

* It auto-configures the bean that is present in the classpath and configures it to run the methods.
* The use of this annotation is reduced in Spring Boot 1.2.0 release because developers provided an alternative of the annotation, i.e. **@SpringBootApplication**.

**@SpringBootApplication:**

* It is a combination of three annotations **@EnableAutoConfiguration, @ComponentScan,** and **@Configuration**.

**Spring MVC and REST Annotations**

**@RequestMapping:**

* It is used to map the **web requests**.
* It has many optional elements like **consumes, header, method, name, params, path, produces**, and **value**.
* We use it with the class as well as the method

.

**Example**

1. @Controller
2. **public** **class** BooksController
3. {
4. @RequestMapping("/computer-science/books")
5. **public** String getAllBooks(Model model)
6. {
7. //application code
8. **return** "bookList";
9. }

**@GetMapping:**

* It maps the **HTTP GET** requests on the specific handler method.
* It is used to create a web service endpoint that **fetches**
* It is used instead of using: **@RequestMapping(method = RequestMethod.GET)**

**@PostMapping:**

* It maps the **HTTP POST**requests on the specific handler method.
* It is used to create a web service endpoint that **creates** It is used instead of using: **@RequestMapping(method = RequestMethod.POST)**

**@PutMapping:**

* It maps the **HTTP PUT** requests on the specific handler method.
* It is used to create a web service endpoint that **creates** or **updates** It is used instead of using: **@RequestMapping(method = RequestMethod.PUT)**

**@DeleteMapping:**

* It maps the **HTTP DELETE** requests on the specific handler method.
* It is used to create a web service endpoint that **deletes**a resource.
* It is used instead of using: **@RequestMapping(method = RequestMethod.DELETE)**

**@PatchMapping:**

* It maps the **HTTP PATCH**requests on the specific handler method.
* It is used instead of using: **@RequestMapping(method = RequestMethod.PATCH)**

**@RequestBody:**

* It is used to **bind** HTTP request with an object in a method parameter.
* Internally it uses **HTTP MessageConverters** to convert the body of the request.
* When we annotate a method parameter with **@RequestBody,** the Spring framework binds the incoming HTTP request body to that parameter.

**@ResponseBody:**

* It binds the method return value to the response body.
* It tells the Spring Boot Framework to serialize a return an object into JSON and XML format

**@PathVariable:**

* It is used to extract the values from the URI.
* It is most suitable for the RESTful web service, where the URL contains a path variable.
* We can define multiple @PathVariable in a method.

**@RequestParam:**

* It is used to extract the query parameters form the URL.
* It is also known as a **query parameter**.
* It is most suitable for web applications.
* It can specify default values if the query parameter is not present in the URL.

**@RequestHeader:**

* It is used to get the details about the HTTP request headers.
* We use this annotation as a **method parameter**.
* The optional elements of the annotation are **name, required, value, defaultValue.**
* For each detail in the header, we should specify separate annotations. We can use it multiple time in a method

**@RestController:**

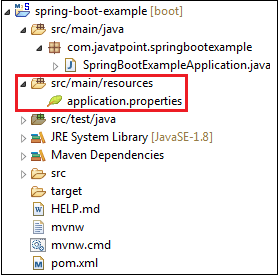
* It can be considered as a combination of **@Controller** and **@ResponseBody**annotations**.**
* The @RestController annotation is itself annotated with the @ResponseBody annotation.
* It eliminates the need for annotating each method with @ResponseBody.

**@RequestAttribute:**

* It binds a method parameter to request attribute.
* It provides convenient access to the request attributes from a controller method.
* With the help of @RequestAttribute annotation, we can access objects that are populated on the server-side.

Spring Boot Application Properties

* Spring Boot Framework comes with a built-in mechanism for application configuration using a file called **application.properties**.
* It is located inside the **src/main/resources** folder, as shown in the following figure.



* Spring Boot provides various properties that can be configured in the **application.properties**file.
* The properties have default values.
* We can set a property(s) for the Spring Boot application. Spring Boot also allows us to define our own property if required.

The application.properties file allows us to run an application in a **different environment.**In short, we can use the application.properties file to:

* Configure the Spring Boot framework
* define our application custom configuration properties

Example of application.properties

1. #configuring application name
2. spring.application.name = demoApplication
3. #configuring port
4. server.port = 8081

Spring Boot Starters

|  |  |
| --- | --- |
| **Name** | **Description** |
| spring-boot-starter-thymeleaf | It is used to build MVC web applications using Thymeleaf views. |
| spring-boot-starter-data-couchbase | It is used for the Couchbase document-oriented database and Spring Data Couchbase. |
| spring-boot-starter-artemis | It is used for JMS messaging using Apache Artemis. |
| spring-boot-starter-web-services | It is used for Spring Web Services. |
| spring-boot-starter-mail | It is used to support Java Mail and Spring Framework's email sending. |
| spring-boot-starter-data-redis | It is used for Redis key-value data store with Spring Data Redis and the Jedis client. |
| spring-boot-starter-web | It is used for building the web application, including RESTful applications using Spring MVC. It uses Tomcat as the default embedded container. |
| spring-boot-starter-data-gemfire | It is used to GemFire distributed data store and Spring Data GemFire. |
| spring-boot-starter-activemq | It is used in JMS messaging using Apache ActiveMQ. |
| spring-boot-starter-data-elasticsearch | It is used in Elasticsearch search and analytics engine and Spring Data Elasticsearch. |
| spring-boot-starter-integration | It is used for Spring Integration. |
| spring-boot-starter-test | It is used to test Spring Boot applications with libraries, including JUnit, Hamcrest, and Mockito. |
| spring-boot-starter-jdbc | It is used for JDBC with the Tomcat JDBC connection pool. |
| spring-boot-starter-mobile | It is used for building web applications using Spring Mobile. |
| spring-boot-starter-validation | It is used for Java Bean Validation with Hibernate Validator. |
| spring-boot-starter-hateoas | It is used to build a hypermedia-based RESTful web application with Spring MVC and Spring HATEOAS. |
| spring-boot-starter-jersey | It is used to build RESTful web applications using JAX-RS and Jersey. An alternative to spring-boot-starter-web. |
| spring-boot-starter-data-neo4j | It is used for the Neo4j graph database and Spring Data Neo4j. |
| spring-boot-starter-data-ldap | It is used for Spring Data LDAP. |
| spring-boot-starter-websocket | It is used for building the WebSocket applications. It uses Spring Framework's WebSocket support. |
| spring-boot-starter-aop | It is used for aspect-oriented programming with Spring AOP and AspectJ. |
| spring-boot-starter-amqp | It is used for Spring AMQP and Rabbit MQ. |
| spring-boot-starter-data-cassandra | It is used for Cassandra distributed database and Spring Data Cassandra. |
| spring-boot-starter-social-facebook | It is used for Spring Social Facebook. |
| spring-boot-starter-jta-atomikos | It is used for JTA transactions using Atomikos. |
| spring-boot-starter-security | It is used for Spring Security. |
| spring-boot-starter-mustache | It is used for building MVC web applications using Mustache views. |
| spring-boot-starter-data-jpa | It is used for Spring Data JPA with Hibernate. |
| spring-boot-starter | It is used for core starter, including auto-configuration support, logging, and YAML. |
| spring-boot-starter-groovy-templates | It is used for building MVC web applications using Groovy Template views. |
| spring-boot-starter-freemarker | It is used for building MVC web applications using FreeMarker views. |
| spring-boot-starter-batch | It is used for Spring Batch. |
| spring-boot-starter-social-linkedin | It is used for Spring Social LinkedIn. |
| spring-boot-starter-cache | It is used for Spring Framework's caching support. |
| spring-boot-starter-data-solr | It is used for the Apache Solr search platform with Spring Data Solr. |
| spring-boot-starter-data-mongodb | It is used for MongoDB document-oriented database and Spring Data MongoDB. |
| spring-boot-starter-jooq | It is used for jOOQ to access SQL databases. An alternative to spring-boot-starter-data-jpa or spring-boot-starter-jdbc. |
| spring-boot-starter-jta-narayana | It is used for Spring Boot Narayana JTA Starter. |
| spring-boot-starter-cloud-connectors | It is used for Spring Cloud Connectors that simplifies connecting to services in cloud platforms like Cloud Foundry and Heroku. |
| spring-boot-starter-jta-bitronix | It is used for JTA transactions using Bitronix. |
| spring-boot-starter-social-twitter | It is used for Spring Social Twitter. |
| spring-boot-starter-data-rest | It is used for exposing Spring Data repositories over REST using Spring Data REST. |

Spring Boot Production Starters

|  |  |
| --- | --- |
| **Name** | **Description** |
| spring-boot-starter-actuator | It is used for Spring Boot's Actuator that provides production-ready features to help you monitor and manage your application. |
| spring-boot-starter-remote-shell | It is used for the CRaSH remote shell to monitor and manage your application over SSH. Deprecated since 1.5. |

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. |

Spring Boot Hello World Example

In the section, we will create a **Maven** project for Hello Word Example. We need the following tools and technologies to develop the same.

* Spring Boot 2.2.2.RELEASE
* JavaSE 1.8
* Maven 3.3.9
* STS IDE

**Step 1:** Open Spring Initializr [https://start.spring.io/](https://start.spring.io/" \t "_blank)

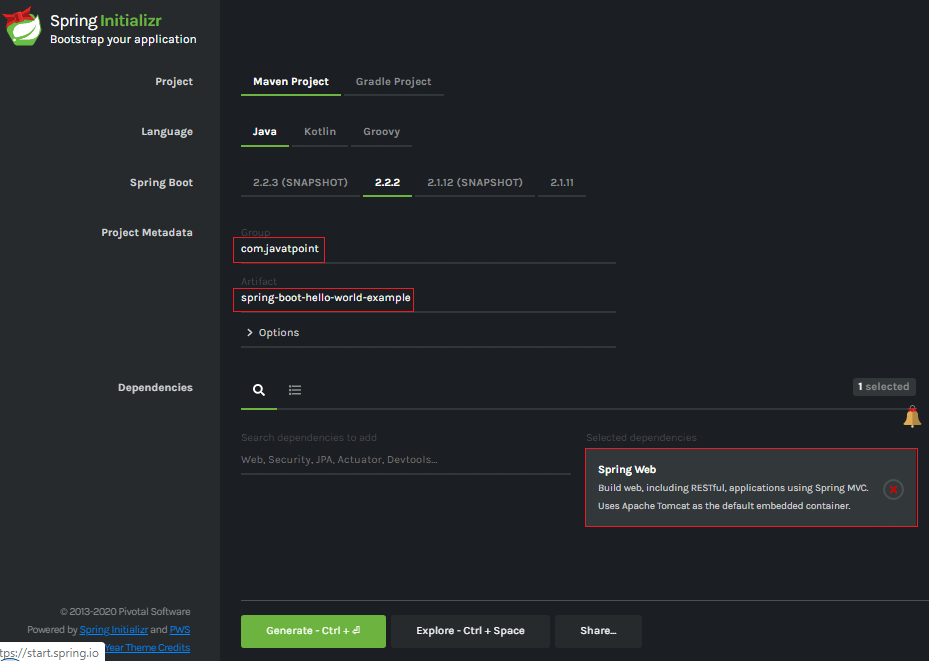
.

**Step 2:** Provide the **Group** name. We have provided **com.javatpoint.**

**Step 3:** Provide the **Artifact** Id. We have provided the **spring-boot-hello-world-example.**

**Step 4:** Add the dependency **Spring Web.**

**Step 5:** Click on the **Generate** button. When we click on the Generate button, it wraps all the specifications into a jar file and downloads it to our local system.

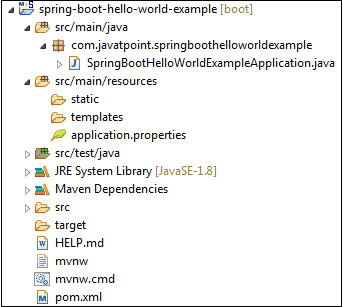


**Step 6: Extract** the RAR file.

**Step 7: Import** the project folder by using the following steps:

File -> Import -> Existing Maven Project -> Next -> Browse -> Select the Project Folder -> Finish

When the project imports successfully, it shows the following project directory in the Package Explorer section of the IDE.



**Step 8:** Create a package with the name **com.javatpoint.controller** inside the folder **src/main/java.**

**Step 9:** Create a Controller class with the name **HelloWorldController.**

**Step 10:** Create a method named **hello()**that returns a String.

**HelloWorldController.java**

1. **package** com.javatpoint.controller;
2. **import** org.springframework.web.bind.annotation.RequestMapping;
3. **import** org.springframework.web.bind.annotation.RestController;
4. @RestController
5. **public** **class** HelloWorldController
6. {
7. @RequestMapping("/")
8. **public** String hello()
9. {
10. **return** "Hello javaTpoint";
11. }
12. }

**Step 11:** Run the **SpringBootHelloWorldExampleApplication.java** file.

**SpringBootHelloWorldExampleApplication.java**

1. **package** com.javatpoint;
2. **import** org.springframework.boot.SpringApplication;
3. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
4. @SpringBootApplication
5. **public** **class** SpringBootHelloWorldExampleApplication
6. {
7. **public** **static** **void** main(String[] args)
8. {
9. SpringApplication.run(SpringBootHelloWorldExampleApplication.**class**, args);
10. }
11. }

When the application runs successfully, it shows a massage in the console, as shown in the following figure.

|  |
| --- |
| Spring Boot Hello World Example  **Step 12:** Open the browser and invoke the URL **https://localhost:8080**. It returns a String that we have specified in the Controller.  Spring Boot Hello World Example |

Spring Boot AOP

* The application is generally developed with multiple layers.
* A typical Java application has the following layers:
* **Web Layer:** It exposes the **services** using the REST or web application.
* **Business Layer:** It implements the **business logic** of an application.
* **Data Layer:** It implements the **persistence logic** of the application.

The responsibility of each layer is different, but there are a few common aspects that apply to all layers are **Logging, Security, validation, caching,** etc. These common aspects are called **cross-cutting concerns.**

If we implement these concerns in each layer separately, the code becomes more difficult to maintain. To overcome this problem, **Aspect-Oriented Programming** (AOP) provides a solution to implement cross-cutting concerns.

* Implement the cross-cutting concern as an aspect.
* Define pointcuts to indicate where the aspect has to be applied.

It ensures that the cross-cutting concerns are defined in one cohesive code component.

AOP

AOP **(Aspect-Oriented Programming)** is a programming pattern that increases modularity by allowing the separation of the **cross-cutting concern**. These cross-cutting concerns are different from the main business logic. We can add additional behavior to existing code without modification of the code itself.

Spring's AOP framework helps us to implement these cross-cutting concerns.

Using AOP, we define common functionality in one place. We are free to define how and where this functionality is applied without modifying the class to which we are applying the new feature. The cross-cutting concern can now be modularized into special classes, called **aspect**.

There are **two** benefits of aspects:

* First, the logic for each concern is now in one place instead of scattered all over the codebase.
* Second, the business modules only contain code for their primary concern. The secondary concern has been moved to the **aspect**.

The aspects have the responsibility that is to be implemented, called **advice**. We can implement an aspect's functionality into a program at one or more join points.

Benefits of AOP

* It is implemented in pure Java.
* There is no requirement for a special compilation process.
* It supports only method execution Join points.
* Only run time weaving is available.
* Two types of AOP proxy is available: **JDK dynamic proxy** and **CGLIB proxy.**

Cross-cutting concern

The cross-cutting concern is a concern that we want to implement in multiple places in an application. It affects the entire application.

AOP Terminology

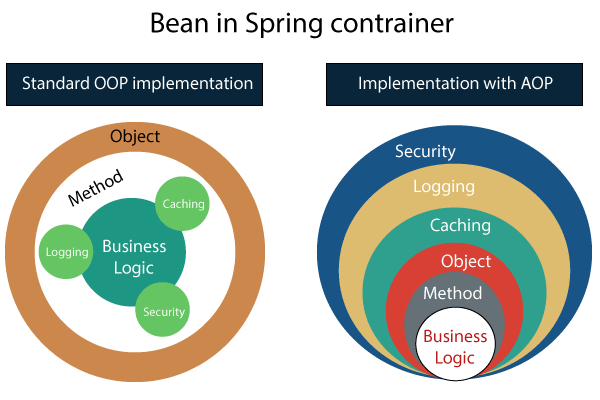
* **Aspect:** An aspect is a module that encapsulates **advice** and **pointcuts** and provides **cross-cutting** An application can have any number of aspects. We can implement an aspect using regular class annotated with **@Aspect** annotation.
* **Pointcut:** A pointcut is an expression that selects one or more join points where advice is executed. We can define pointcuts using **expressions** or **patterns**. It uses different kinds of expressions that matched with the join points. In Spring Framework, **AspectJ** pointcut expression language is used.
* **Join point:** A join point is a point in the application where we apply an **AOP aspect**. Or it is a specific execution instance of an advice. In AOP, join point can be a **method execution, exception handling, changing object variable value**, etc.
* **Advice:** The advice is an action that we take either **before** or **after** the method execution. The action is a piece of code that invokes during the program execution. There are **five** types of advices in the Spring AOP framework: **before, after, after-returning, after-throwing,**and **around advice.**Advices are taken for a particular **join point.**We will discuss these advices further in this section.
* **Target object:** An object on which advices are applied, is called the **target object**. Target objects are always a **proxied** It means a subclass is created at run time in which the target method is overridden, and advices are included based on their configuration.
* **Weaving:** It is a process of **linking aspects** with other application types. We can perform weaving at **run time, load time,** and **compile time**.

**Proxy:** It is an object that is created after applying advice to a target object is called **proxy**. The Spring AOP implements the **JDK dynamic proxy** to create the proxy classes with target classes and advice invocations. These are called AOP proxy classes.

AOP vs. OOP

The differences between AOP and OOP are as follows:

|  |  |
| --- | --- |
| **AOP** | **OOP** |
| **Aspect:** A code unit that encapsulates pointcuts, advices, and attributes. | **Class:** A code unit that encapsulates methods and attributes. |
| **Pointcut:** It defines the set of entry points in which advice is executed. | **Method signature:** It defines the entry points for the execution of method bodies. |
| **Advice:** It is an implementation of cross-cutting concerns. | **Method bodies:** It is an implementation of the business logic concerns. |
| **Waver: It constructs code (source or object) with advice.** | **Compiler: It converts source code to object code.** |



Spring AOP vs. AspectJ

The differences between AOP and OOP are as follows:

|  |  |
| --- | --- |
| **Spring AOP** | **AspectJ** |
| There is a need for a separate compilation process. | It requires the AspectJ compiler. |
| It supports only method execution pointcuts. | It supports all pointcuts. |
| It can be implemented on beans managed by Spring Container. | It can be implemented on all domain objects. |
| It supports only method level weaving. | It can wave fields, methods, constructors, static initializers, final class, etc. |

Types of AOP Advices

There are five types of AOP advices are as follows:

* Before Advice
* After Advice
* Around Advice
* After Throwing
* After Returning

**Before Advice:**

* An advice that executes before a join point, is called before advice.
* We use **@Before** annotation to mark an advice as Before advice.

**After Advice:**

* An advice that executes after a join point, is called after advice.
* We use **@After**annotation to mark an advice as After advice.

**Around Advice:**

* An advice that executes before and after of a join point, is called around advice.

**After Throwing Advice:**

* An advice that executes when a join point throws an exception.

**After Returning Advice:**

* An advice that executes when a method executes successfully.

Before implementing the AOP in an application, we are required to add **Spring AOP** dependency in the pom.xml file.

## Spring Boot Starter AOP

Spring Boot Starter AOP is a dependency that provides Spring AOP and AspectJ. Where AOP provides basic AOP capabilities while the AspectJ provides a complete AOP framework.

1. **<dependency>**
2. **<groupId>**org.springframework.boot**</groupId>**
3. **<artifactId>**spring-boot-starter-aop**</artifactId>**
4. **<version>**2.2.2.RELEASE**</version>**
5. **</dependency>**

In the next section, we will implement the different advices in the application.

Spring Boot AOP Before Advice

* Before advice is used in Aspect-Oriented Programming to achieve the cross-cutting.
* It is an advice type which ensures that an advice runs before the method execution.
* We use **@Before** annotation to implement the before advice.

Let's understand before advice through an example.

Spring Boot Before Advice Example

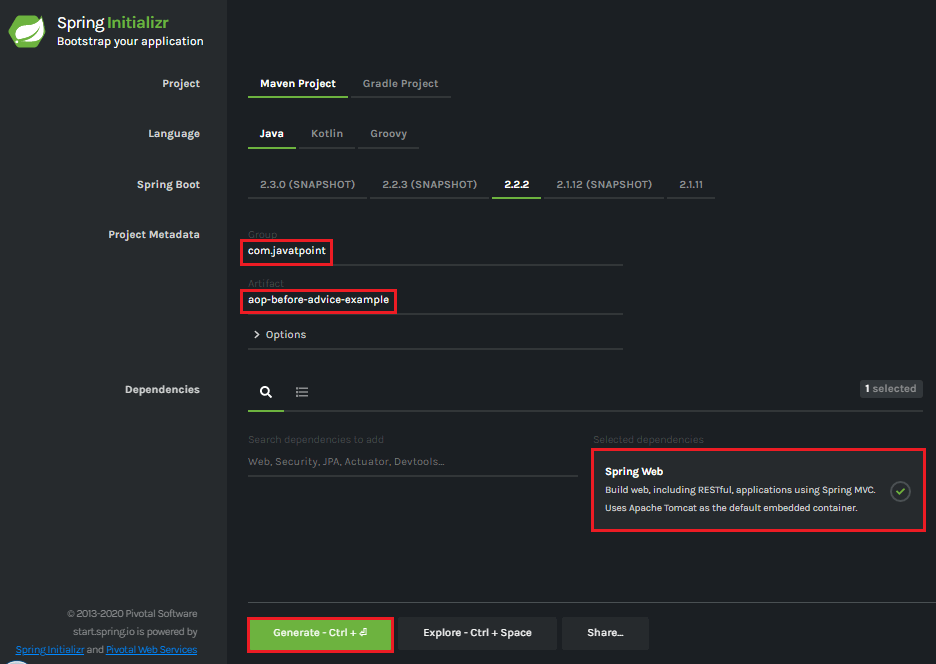
**Step 1:** Open Spring Initializr [http://start.spring.io](https://start.spring.io/).

**Step 2:** Provide the **Group** name. We have provided the Group name **com.javatpoint.**

**Step 3:** Provide the **Artifact Id.** We have provided the Artifact Id **aop-before-advice-example.**

**Step 4:** Add the **Spring Web**dependency.

**Step 5:** Click on the **Generate** button. When we click on the Generate button, it wraps all the specifications in a **jar** file and downloads it to the local system.



**Step 6: Extract** the downloaded jar file.

**Step 7: Import** the folder by using the following steps:

File -> Import -> Existing Maven Projects -> Next -> Browse the Folder **aop-before-advice-example**-> Finish.

**Step 8:** Open the **pom.xml**file and add the following **AOP** dependency. It is a starter for aspect-oriented programming with **Spring AOP** and **AspectJ**.

1. **<dependency>**
2. **<groupId>**org.springframework.boot**</groupId>**
3. **<artifactId>**spring-boot-starter-aop**</artifactId>**
4. **</dependency>**
5. **</dependencies>**

**pom.xml**

1. **<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"**>**
2. **<modelVersion>**4.0.0**</modelVersion>**
3. **<groupId>**com.javatpoint**</groupId>**
4. **<artifactId>** aop-before-advice-example**</artifactId>**
5. **<version>**0.0.1-SNAPSHOT**</version>**
6. **<packaging>**jar**</packaging>**
7. **<name>**aop-before-advice-example**</name>**
8. **<description>**Demo project for Spring Boot**</description>**
9. **<parent>**
10. **<groupId>**org.springframework.boot**</groupId>**
11. **<artifactId>**spring-boot-starter-parent**</artifactId>**
12. **<version>**2.2.2.RELEASE**</version>**
13. **<relativePath** **/>** <!-- lookup parent from repository -->
14. **</parent>**
15. **<properties>**
16. **<project.build.sourceEncoding>**UTF-8**</project.build.sourceEncoding>**
17. **<project.reporting.outputEncoding>**UTF-8**</project.reporting.outputEncoding>**
18. **<java.version>**1.8**</java.version>**
19. **</properties>**
20. **<dependencies>**
21. **<dependency>**
22. **<groupId>**org.springframework.boot**</groupId>**
23. **<artifactId>**spring-boot-starter-web**</artifactId>**
24. **</dependency>**
25. **<dependency>**
26. **<groupId>**org.springframework.boot**</groupId>**
27. **<artifactId>**spring-boot-starter-aop**</artifactId>**
28. **</dependency>**
29. **</dependencies>**
31. **<build>**
32. **<plugins>**
33. **<plugin>**
34. **<groupId>**org.springframework.boot**</groupId>**
35. **<artifactId>**spring-boot-maven-plugin**</artifactId>**
36. **</plugin>**
37. **</plugins>**
38. **</build>**
39. **</project>**

**Step 9:** Open **AopBeforeAdviceExampleApplication.java** file and add an annotation **@EnableAspectJAutoProxy.**

1. @EnableAspectJAutoProxy(proxyTargetClass=true)

It enables support for handling components marked with AspectJ’s @Aspect annotation. It is used with @Configuration annotation. We can control the type of proxy by using the **proxyTargetClass** attribute. Its default value is **false**.

**AopBeforeAdviceExampleApplication.java**

1. **package** com.javatpoint;
2. **import** org.springframework.boot.SpringApplication;
3. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
4. **import** org.springframework.context.annotation.EnableAspectJAutoProxy;
5. @SpringBootApplication
6. @EnableAspectJAutoProxy(proxyTargetClass=**true**)
7. **public** **class** AopBeforeAdviceExampleApplication
8. {
9. **public** **static** **void** main(String[] args) {
10. SpringApplication.run(AopBeforeAdviceExampleApplication.**class**, args);
11. }
12. }

**Step 10:** Create a package with the name **com.javatpoint.model.**

**Step 11:** Create a model class under the package **com.javatpoint.model.**We have created a class with the name **Employee.**In the class, define the following:

* Define three variables**empId, firstName,**and**secondName**of type String.
* Generate **Getters and Setters.**
* Create a **default**

**Employee.java**

1. **package** com.javatpoint.model;
2. **public** **class** Employee
3. {
4. **private** String empId;
5. **private** String firstName;
6. **private** String secondName;
7. //default constructor
8. **public** Employee()
9. {
10. }
11. **public** String getEmpId()
12. {
13. **return** empId;
14. }
15. **public** **void** setEmpId(String empId)
16. {
17. **this**.empId = empId;
18. }
19. **public** String getFirstName()
20. {
21. **return** firstName;
22. }
23. **public** **void** setFirstName(String firstName)
24. {
25. **this**.firstName = firstName;
26. }
27. **public** String getSecondName()
28. {
29. **return** secondName;
30. }
31. **public** **void** setSecondName(String secondName)
32. {
33. **this**.secondName = secondName;
34. }
35. }

**Step 12:** Create a package with the name **com.javatpoint.controller.**

**Step 13:** Create a controller class under the package **com.javatpoint.controller.**We have created a class with the name **EmployeeController.**

In the controller class, we have defined the two mappings one for adding an employee and the other for removing an employee.

**EmployeeController.java**

1. **package** com.javatpoint.controller;
2. **import** org.springframework.beans.factory.annotation.Autowired;
3. **import** org.springframework.web.bind.annotation.RequestMapping;
4. **import** org.springframework.web.bind.annotation.RequestMethod;
5. **import** org.springframework.web.bind.annotation.RequestParam;
6. **import** org.springframework.web.bind.annotation.RestController;
7. **import** com.javatpoint.model.Employee;
8. **import** com.javatpoint.service.EmployeeService;
9. @RestController
10. **public** **class** EmployeeController
11. {
12. @Autowired
13. **private** EmployeeService employeeService;
14. @RequestMapping(value = "/add/employee", method = RequestMethod.GET)
15. **public** com.javatpoint.model.Employee addEmployee(@RequestParam("empId") String empId, @RequestParam("firstName") String firstName, @RequestParam("secondName") String secondName)
16. {
17. **return** employeeService.createEmployee(empId, firstName, secondName);
18. }
19. @RequestMapping(value = "/remove/employee", method = RequestMethod.GET)
20. **public** String removeEmployee( @RequestParam("empId") String empId)
21. {
22. employeeService.deleteEmployee(empId);
23. **return** "Employee removed";
24. }
25. }

**Step 14:** Create a package with the name **com.javatpoint.service.**

**Step 15:** Create a Service class under the package **com.javatpoint.service.**We have created a class with the name **EmployeeService.**

In the Service class, we have defined two methods **createEmployee**and**deleteEmployee.**

**EmployeeService.java**

1. **package** com.javatpoint.service;
2. **import** org.springframework.stereotype.Service;
3. **import** com.javatpoint.model.Employee;
4. @Service
5. **public** **class** EmployeeService
6. {
7. **public** Employee createEmployee( String empId, String fname, String sname)
8. {
9. Employee emp = **new** Employee();
10. emp.setEmpId(empId);
11. emp.setFirstName(fname);
12. emp.setSecondName(sname);
13. **return** emp;
14. }
15. **public** **void** deleteEmployee(String empId)
16. {
17. }
18. }

**Step 16:** Create a package with the name **com.javatpoint.aspect.**

**Step 17:** Create an aspect class under the package **com.javatpoint.aspect.**We have created a class with the name **EmployeeServiceAspect.**

In the aspect class, we have defined the before advice logic.

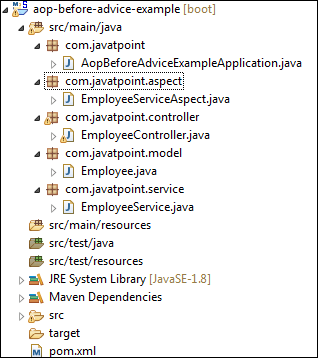
**EmployeeServiceAspect.java**

1. **package** com.javatpoint.aspect;
2. **import** org.aspectj.lang.JoinPoint;
3. **import** org.aspectj.lang.annotation.Aspect;
4. **import** org.aspectj.lang.annotation.Before;
5. **import** org.springframework.stereotype.Component;
6. @Aspect
7. @Component
8. **public** **class** EmployeeServiceAspect
9. {
10. @Before(value = "execution(\* com.javatpoint.service.EmployeeService.\*(..)) and args(empId, fname, sname)")
11. **public** **void** beforeAdvice(JoinPoint joinPoint, String empId, String fname, String sname) {
12. System.out.println("Before method:" + joinPoint.getSignature());
13. System.out.println("Creating Employee with first name - " + fname + ", second name - " + sname + " and id - " + empId);
14. }
15. }

In the above class:

* **execution(expression):** The expression is a method on which advice is to be applied.
* **@Before:** It marks a function as an advice to be executed before method that covered by PointCut.

After creating all the modules, the project directory looks like the following:

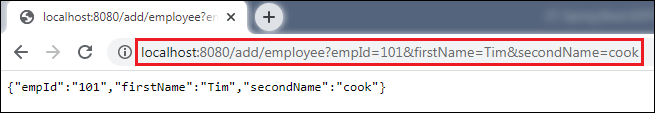


We have set-up all the modules. Now we will run the application.

**Step 18:** Open the e**AopBeforeAdviceExampleApplication.java** file and run it as Java Application.

**Step 19:** Open the browser and invoke the following URL : *http://localhost:8080/add/employee?empId={id}&firstName={fname}&secondName={sname}*

In the above URL, **/add/employee**is the mapping that we have created in the Controller class. We have used two separators **(?)**and **(&)**for separating two values.

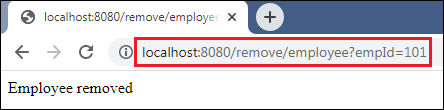


In the above output, we have assigned **emId 101, firstName=Tim,**and**secondName=cook.**

Let's have a look at the console. We see that before invoking the **createEmployee**() method of **EmployeeService** class, the method **beforeAdvice()** of **EmployeeServiceAspect** class invokes, as shown below.

Spring Boot AOP Before Advice

Similarly, we can also remove an employee by invoking the URL http://localhost:8080/remove/employee?empId=101. It returns a message **Employee removed**, as shown in the following figure.



In this section, we have learned the working of before advice. In the next section, we will learn the working of after advice and implement it in an apllication.

**Spring Boot AOP After Advice**

After advice is used in Aspect-Oriented Programming to achieve the cross-cutting. It is an advice type which ensures that an advice runs after the method execution. We use **@After** annotation to implement the after advice.

Let's understand after advice through an example.

Spring Boot After Advice Example

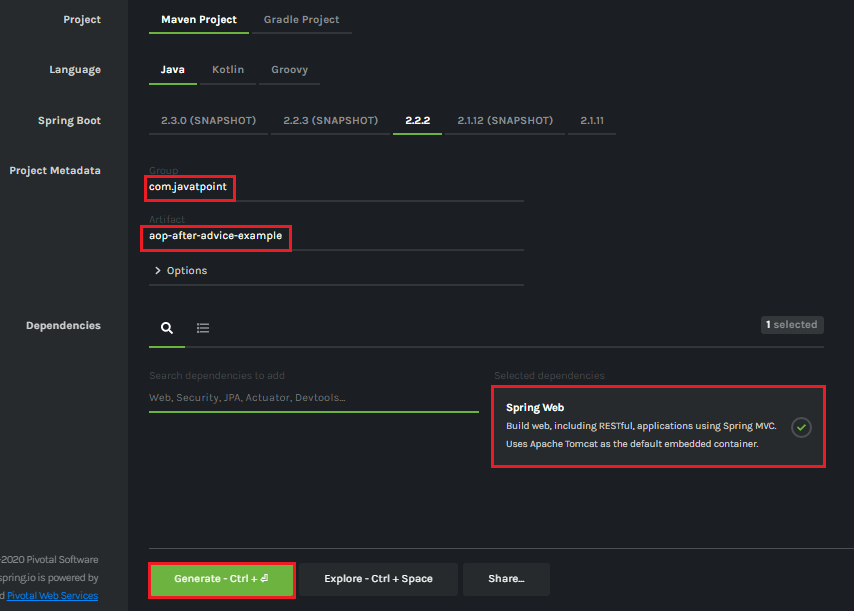
**Step 1:** Open Spring Initializr [http://start.spring.io](https://start.spring.io/).

**Step 2:** Provide the **Group** name. We have provided the Group name **com.javatpoint.**

**Step 3:** Provide the **Artifact Id.** We have provided the Artifact Id **aop-after-advice-example.**

**Step 4:** Add the **Spring Web** dependency.

**Step 5:** Click on the **Generate** button. When we click on the Generate button, it wraps all the specifications in a **jar** file and downloads it to the local system.



**Step 6: Extract** the downloaded jar file.

**Step 7: Import** the folder by using the following steps:

File -> Import -> Existing Maven Projects -> Next -> Browse the Folder **aop-after-advice-example**-> Finish.

**Step 8:** Open the **pom.xml**file and add the following **AOP** dependency. It is a starter for aspect-oriented programming with **Spring AOP** and **AspectJ**.

1. **<dependency>**
2. **<groupId>**org.springframework.boot**</groupId>**
3. **<artifactId>**spring-boot-starter-aop**</artifactId>**
4. **</dependency>**
5. **</dependencies>**

**pom.xml**

1. **<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"**>**
2. **<modelVersion>**4.0.0**</modelVersion>**
3. **<groupId>**com.javatpoint**</groupId>**
4. **<artifactId>** aop-after-advice-example**</artifactId>**
5. **<version>**0.0.1-SNAPSHOT**</version>**
6. **<packaging>**jar**</packaging>**
7. **<name>**aop-after-advice-example**</name>**
8. **<description>**Demo project for Spring Boot**</description>**
9. **<parent>**
10. **<groupId>**org.springframework.boot**</groupId>**
11. **<artifactId>**spring-boot-starter-parent**</artifactId>**
12. **<version>**2.2.2.RELEASE**</version>**
13. **<relativePath** **/>** <!-- lookup parent from repository -->
14. **</parent>**
15. **<properties>**
16. **<project.build.sourceEncoding>**UTF-8**</project.build.sourceEncoding>**
17. **<project.reporting.outputEncoding>**UTF-8**</project.reporting.outputEncoding>**
18. **<java.version>**1.8**</java.version>**
19. **</properties>**
20. **<dependencies>**
21. **<dependency>**
22. **<groupId>**org.springframework.boot**</groupId>**
23. **<artifactId>**spring-boot-starter-web**</artifactId>**
24. **</dependency>**
25. **<dependency>**
26. **<groupId>**org.springframework.boot**</groupId>**
27. **<artifactId>**spring-boot-starter-aop**</artifactId>**
28. **</dependency>**
29. **</dependencies>**
31. **<build>**
32. **<plugins>**
33. **<plugin>**
34. **<groupId>**org.springframework.boot**</groupId>**
35. **<artifactId>**spring-boot-maven-plugin**</artifactId>**
36. **</plugin>**
37. **</plugins>**
38. **</build>**
39. **</project>**

**Step 9:** Open **AopAfterAdviceExampleApplication.java** file and add an annotation **@EnableAspectJAutoProxy.**

1. @EnableAspectJAutoProxy(proxyTargetClass=true)

It enables support for handling components marked with AspectJ's **@Aspect** annotation. It is used with @Configuration annotation. We can control the type of proxy by using the **proxyTargetClass** attribute. Its default value is **false**.

**AopAfterAdviceExampleApplication.java**

1. **package** com.javatpoint;
2. **import** org.springframework.boot.SpringApplication;
3. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
4. **import** org.springframework.context.annotation.EnableAspectJAutoProxy;
5. @SpringBootApplication
6. @EnableAspectJAutoProxy(proxyTargetClass=**true**)
7. **public** **class** AopAfterAdviceExampleApplication
8. {
9. **public** **static** **void** main(String[] args) {
10. SpringApplication.run(AopAfterAdviceExampleApplication.**class**, args);
11. }
12. }

**Step 10:** Create a package with the name **com.javatpoint.model.**

**Step 11:** Create a model class under the package **com.javatpoint.model.**We have created a class with the name **Employee.**In the class, define the following:

* Define three variables**empId, firstName,**and**secondName**of type String.
* Generate **Getters and Setters.**
* Create a **default**

**Employee.java**

1. **package** com.javatpoint.model;
2. **public** **class** Employee
3. {
4. **private** String empId;
5. **private** String firstName;
6. **private** String secondName;
7. //default constructor
8. **public** Employee()
9. {
10. }
11. **public** String getEmpId()
12. {
13. **return** empId;
14. }
15. **public** **void** setEmpId(String empId)
16. {
17. **this**.empId = empId;
18. }
19. **public** String getFirstName()
20. {
21. **return** firstName;
22. }
23. **public** **void** setFirstName(String firstName)
24. {
25. **this**.firstName = firstName;
26. }
27. **public** String getSecondName()
28. {
29. **return** secondName;
30. }
31. **public** **void** setSecondName(String secondName)
32. {
33. **this**.secondName = secondName;
34. }
35. }

**Step 12:** Create a package with the name **com.javatpoint.controller.**

**Step 13:** Create a controller class under the package **com.javatpoint.controller.**We have created a class with the name **EmployeeController.**

In the controller class, we have defined the two mappings one for adding an employee and the other for removing an employee.

**EmployeeController.java**

1. **package** com.javatpoint.controller;
2. **import** org.springframework.beans.factory.annotation.Autowired;
3. **import** org.springframework.web.bind.annotation.RequestMapping;
4. **import** org.springframework.web.bind.annotation.RequestMethod;
5. **import** org.springframework.web.bind.annotation.RequestParam;
6. **import** org.springframework.web.bind.annotation.RestController;
7. **import** com.javatpoint.model.Employee;
8. **import** com.javatpoint.service.EmployeeService;
9. @RestController
10. **public** **class** EmployeeController
11. {
12. @Autowired
13. **private** EmployeeService employeeService;
14. @RequestMapping(value = "/add/employee", method = RequestMethod.GET)
15. **public** com.javatpoint.model.Employee addEmployee(@RequestParam("empId") String empId, @RequestParam("firstName") String firstName, @RequestParam("secondName") String secondName)
16. {
17. **return** employeeService.createEmployee(empId, firstName, secondName);
18. }
19. @RequestMapping(value = "/remove/employee", method = RequestMethod.GET)
20. **public** String removeEmployee( @RequestParam("empId") String empId)
21. {
22. employeeService.deleteEmployee(empId);
23. **return** "Employee removed";
24. }
25. }

**Step 14:** Create a package with the name **com.javatpoint.service.**

**Step 15:** Create a Service class under the package **com.javatpoint.service.**We have created a class with the name **EmployeeService.**

In the Service class, we have defined two methods **createEmployee**and**deleteEmployee.**

**EmployeeService.java**

1. **package** com.javatpoint.service;
2. **import** org.springframework.stereotype.Service;
3. **import** com.javatpoint.model.Employee;
4. @Service
5. **public** **class** EmployeeService
6. {
7. **public** Employee createEmployee( String empId, String fname, String sname)
8. {
9. Employee emp = **new** Employee();
10. emp.setEmpId(empId);
11. emp.setFirstName(fname);
12. emp.setSecondName(sname);
13. **return** emp;
14. }
15. **public** **void** deleteEmployee(String empId)
16. {
17. }
18. }

**Step 16:** Create a package with the name **com.javatpoint.aspect.**

**Step 17:** Create an aspect class under the package **com.javatpoint.aspect.**We have created a class with the name **EmployeeServiceAspect.**

In the aspect class, we have defined the after-advice logic.

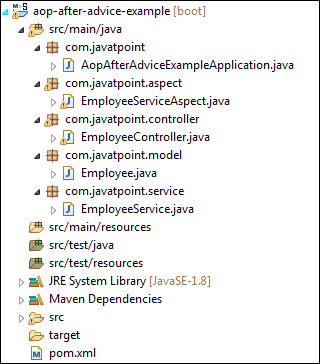
**EmployeeServiceAspect.java**

1. package com.javatpoint.aspect;
2. import org.aspectj.lang.JoinPoint;
3. import org.aspectj.lang.annotation.Aspect;
4. import org.aspectj.lang.annotation.After;
5. import org.springframework.stereotype.Component;
6. @Aspect
7. @Component
8. public class EmployeeServiceAspect
9. {
10. @After(value = "execution(\* com.javatpoint.service.EmployeeService.\*(..)) and args(empId, fname, sname)")
11. public void afterAdvice(JoinPoint joinPoint, String empId, String fname, String sname) {
12. System.out.println("After method:" + joinPoint.getSignature());
13. System.out.println("Creating Employee with first name - " + fname + ", second name - " + sname + " and id - " + empId);
14. }
15. }

In the above class:

* **execution(expression):** The expression is a method on which advice is to be applied.
* **@After:** The method annotated with **@After** executes after all the methods that matched with the pointcut expression.

After creating all the modules, the project directory looks like the following:

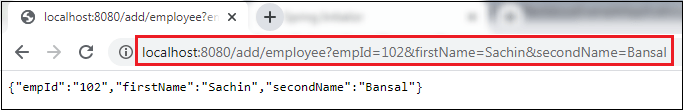


We have set-up all the modules. Now we will run the application.

**Step 18:** Open the **AopAfterAdviceExampleApplication.java** file and run it as Java Application.

**Step 19:** Open the browser and invoke the following URL : *http://localhost:8080/add/employee?empId={id}&firstName={fname}&secondName={sname}*

In the above URL, **/add/employee**is the mapping that we have created in the Controller class. We have used two separators **(?)**and **(&)**for separating two values.

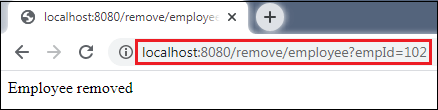


In the above output, we have assigned **emId 102, firstName=Sachin,**and**secondName=Bansal.**

Let's have a look at the console. We see that after invoking the **createEmployee**() method of **EmployeeService** class, the method **afterAdvice()** of **EmployeeServiceAspect** class invokes, as shown below.

Spring Boot AOP After Advice

Similarly, we can also remove an employee by invoking the URL http://localhost:8080/remove/employee?empId=102. It returns a message **Employee removed**, as shown in the following figure.



In this section, we have learned the working of after advice. In the next section, we will learn the working of around advice.

Spring Boot JPA

What is JPA?

**Spring Boot JPA**is a Java specification for managing **relational** data in Java applications. It allows us to access and persist data between Java object/ class and relational database. JPA follows **Object-Relation Mapping**(ORM). It is a set of interfaces. It also provides a runtime **EntityManager** API for processing queries and transactions on the objects against the database. It uses a platform-independent object-oriented query language JPQL (Java Persistent Query Language).

In the context of persistence, it covers three areas:

* The Java Persistence API
* **Object-Relational** metadata
* The API itself, defined in the **persistence** package

JPA is not a framework. It defines a concept that can be implemented by any framework.

Why should we use JPA?

JPA is simpler, cleaner, and less labor-intensive than JDBC, SQL, and hand-written mapping. JPA is suitable for non-performance oriented complex applications. The main advantage of JPA over JDBC is that, in JPA, data is represented by objects and classes while in JDBC data is represented by tables and records. It uses POJO to represent persistent data that simplifies database programming. There are some other advantages of JPA:

* JPA avoids writing DDL in a database-specific dialect of SQL. Instead of this, it allows mapping in XML or using Java annotations.
* JPA allows us to avoid writing DML in the database-specific dialect of SQL.
* JPA allows us to save and load Java objects and graphs without any DML language at all.
* When we need to perform queries JPQL, it allows us to express the queries in terms of Java entities rather than the (native) SQL table and columns.

JPA Features

There are following features of JPA:

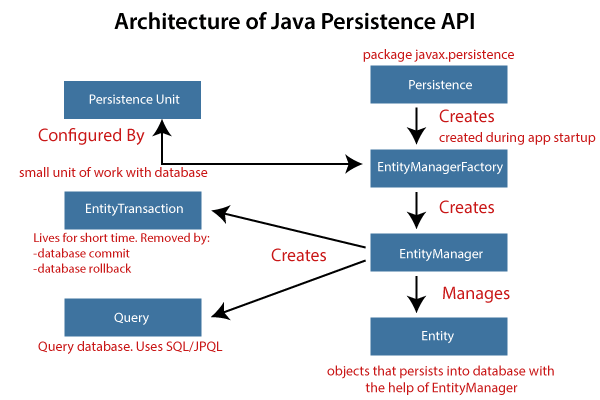
* It is a powerful repository and custom **object-mapping abstraction.**
* It supports for **cross-store persistence**. It means an entity can be partially stored in MySQL and Neo4j (Graph Database Management System).
* It dynamically generates queries from queries methods name.
* The domain base classes provide basic properties.
* It supports transparent auditing.
* Possibility to integrate custom repository code.
* It is easy to integrate with Spring Framework with the custom namespace.

JPA Architecture

JPA is a source to store business entities as relational entities. It shows how to define a POJO as an entity and how to manage entities with relation.

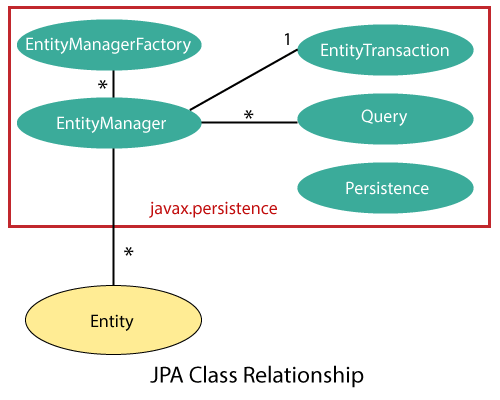
The following figure describes the class-level architecture of JPA that describes the core classes and interfaces of JPA that is defined in the **javax persistence** package. The JPA architecture contains the following units:

* **Persistence:** It is a class that contains static methods to obtain an EntityManagerFactory instance.
* **EntityManagerFactory:** It is a factory class of EntityManager. It creates and manages multiple instances of EntityManager.
* **EntityManager:** It is an interface. It controls the persistence operations on objects. It works for the Query instance.
* **Entity:** The entities are the persistence objects stores as a record in the database.
* **Persistence Unit:** It defines a set of all entity classes. In an application, EntityManager instances manage it. The set of entity classes represents the data contained within a single data store.
* **EntityTransaction:** It has a **one-to-one** relationship with the EntityManager class. For each EntityManager, operations are maintained by EntityTransaction class.
* **Query:** It is an interface that is implemented by each JPA vendor to obtain relation objects that meet the criteria.



JPA Class Relationships

The classes and interfaces that we have discussed above maintain a relationship. The following figure shows the relationship between classes and interfaces.



* The relationship between EntityManager and EntiyTransaction is **one-to-one**. There is an EntityTransaction instance for each EntityManager operation.
* The relationship between EntityManageFactory and EntiyManager is **one-to-many**. It is a factory class to EntityManager instance.
* The relationship between EntityManager and Query is **one-to-many**. We can execute any number of queries by using an instance of EntityManager class.
* The relationship between EntityManager and Entity is **one-to-many**. An EntityManager instance can manage multiple Entities.

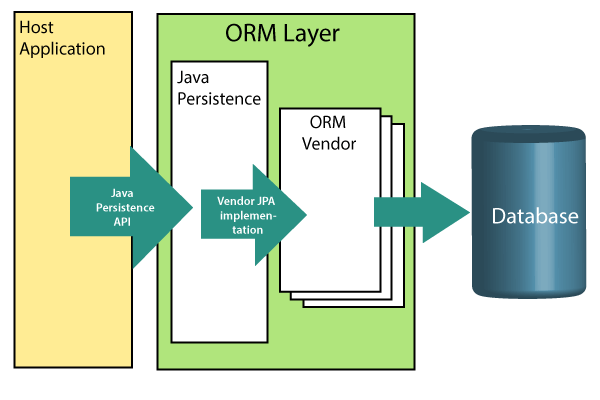
JPA Implementations

JPA is an open-source API. There is various enterprises vendor such as Eclipse, RedHat, Oracle, etc. that provides new products by adding the JPA in them. There are some popular JPA implementations frameworks such as **Hibernate, EclipseLink, DataNucleus,** etc. It is also known as **Object-Relation Mapping** (ORM) tool.

Object-Relation Mapping (ORM)

In ORM, the mapping of Java objects to database tables, and vice-versa is called **Object-Relational Mapping.** The ORM mapping works as a bridge between a **relational database** (tables and records) and **Java application** (classes and objects).

In the following figure, the ORM layer is an adapter layer. It adapts the language of object graphs to the language of SQL and relation tables.



The ORM layer exists between the application and the database. It converts the Java classes and objects so that they can be stored and managed in a relational database. By default, the name that persists become the name of the table, and fields become columns. Once an application sets-up, each table row corresponds to an object.

Spring Boot CRUD Operations

What is the CRUD operation?

The **CRUD** stands for **Create, Read/Retrieve, Update,** and **Delete**. These are the four basic functions of the persistence storage.

The CRUD operation can be defined as user interface conventions that allow view, search, and modify information through computer-based forms and reports. CRUD is data-oriented and the standardized use of **HTTP action verbs**. HTTP has a few important verbs.

* **POST:** Creates a new resource
* **GET:** Reads a resource
* **PUT:** Updates an existing resource
* **DELETE:** Deletes a resource

Within a database, each of these operations maps directly to a series of commands. However, their relationship with a RESTful API is slightly more complex.

Standard CRUD Operation

* **CREATE Operation:** It performs the INSERT statement to create a new record.
* **READ Operation:** It reads table records based on the input parameter.
* **UPDATE Operation:** It executes an update statement on the table. It is based on the input parameter.
* **DELETE Operation:** It deletes a specified row in the table. It is also based on the input parameter.

How CRUD Operations Works

CRUD operations are at the foundation of the most dynamic websites. Therefore, we should differentiate **CRUD** from the **HTTP** **action verbs**.

Suppose, if we want to **create** a new record, we should use HTTP action verb **POST**. To **update** a record, we should use the **PUT** verb. Similarly, if we want to **delete** a record, we should use the **DELETE**verb. Through CRUD operations, users and administrators have the right to retrieve, create, edit, and delete records online.

We have many options for executing CRUD operations. One of the most efficient choices is to create a set of stored procedures in SQL to execute operations.

The CRUD operations refer to all major functions that are implemented in relational database applications. Each letter of the CRUD can map to a SQL statement and HTTP methods.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **SQL** | **HTTP verbs** | **RESTful Web Service** |
| **Create** | INSERT | PUT/POST | POST |
| **Read** | SELECT | GET | GET |
| **Update** | UPDATE | PUT/POST/PATCH | PUT |
| **Delete** | DELETE | DELETE | DELETE |

Spring Boot CrudRepository

* Spring Boot provides an interface called **CrudRepository**that contains methods for CRUD operations.
* It is defined in the package **org.springframework.data.repository**.
* It extends the Spring Data**Repository** interface.
* It provides generic Crud operation on a repository.
* If we want to use CrudRepository in an application, we have to create an interface and extend the **CrudRepository**.

**Syntax**

1. **public** **interface** CrudRepository<T,ID> **extends** Repository<T,ID>

where,

* **T** is the domain type that repository manages.
* **ID** is the type of the id of the entity that repository manages.

For example:

1. **public** **interface** StudentRepository **extends** CrudRepository<Student, Integer>
2. {
3. }

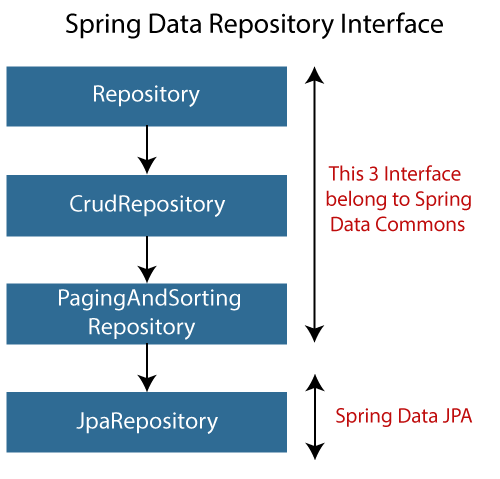
In the above example, we have created an interface named **StudentRepository** that extends CrudRepository. Where **Student** is the repository to manage, and **Integer** is the type of Id that is defined in the Student repository.

Spring Boot JpaRepository

JpaRepository provides JPA related methods such as flushing, persistence context, and deletes a record in a batch. It is defined in the package **org.springframework.data.jpa.repository.**JpaRepository extends both **CrudRepository** and **PagingAndSortingRepository.**

For example:

1. **public** **interface** BookDAO **extends** JpaRepository
2. {
3. }



Why should we use these interfaces?

* The interfaces allow Spring to find the repository interface and create proxy objects for that.
* It provides methods that allow us to perform some common operations. We can also define custom methods as well.

CrudRepository vs. JpaRepository

|  |  |
| --- | --- |
| **CrudRepository** | **JpaRepository** |
| CrudRepository does not provide any method for pagination and sorting. | JpaRepository extends PagingAndSortingRepository. It provides all the methods for implementing the pagination. |
| It works as a **marker** interface. | JpaRepository extends both **CrudRepository** and **PagingAndSortingRepository**. |
| It provides CRUD function only. For example **findById(), findAll(),** etc. | It provides some extra methods along with the method of PagingAndSortingRepository and CrudRepository. For example, **flush(), deleteInBatch().** |
| It is used when we do not need the functions provided by JpaRepository and PagingAndSortingRepository. | It is used when we want to implement pagination and sorting functionality in an application. |

Spring Boot CRUD Operation Example

Let's set up a Spring Boot application and perform CRUD operation.

**Step 1:** Open Spring Initializr [http://start.spring.io](https://start.spring.io/).

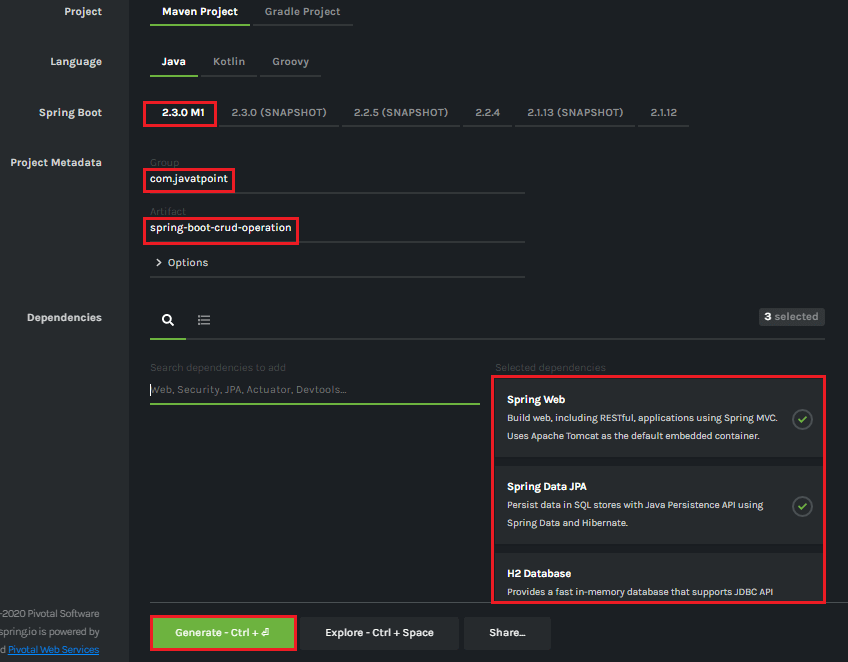
**Step 2:** Select the Spring Boot version **2.3.0.M1.**

**Step 2:** Provide the **Group** name. We have provided **com.javatpoint.**

**Step 3:** Provide the **Artifact** Id. We have provided **spring-boot-crud-operation.**

**Step 5:** Add the dependencies **Spring Web, Spring Data JPA,**and**H2 Database.**

**Step 6:** Click on the **Generate** button. When we click on the Generate button, it wraps the specifications in a **Jar** file and downloads it to the local system.



**Step 7: Extract** the Jar file and paste it into the STS workspace.

**Step 8: Import** the project folder into STS.

File -> Import -> Existing Maven Projects -> Browse -> Select the folder spring-boot-crud-operation -> Finish

It takes some time to import.

**Step 9:** Create a package with the name **com.javatpoint.model**in the folder **src/main/java.**

**Step 10:** Create a model class in the package **com.javatpoint.model.**We have created a model class with the name **Books.**In the Books class, we have done the following:

* Define four variable **bookid, bookname, author,**and
* Generate Getters and Setters.  
  Right-click on the file -> Source -> Generate Getters and Setters.
* Mark the class as an **Entity** by using the annotation **@Entity.**
* Mark the class as **Table** name by using the annotation **@Table.**
* Define each variable as **Column** by using the annotation **@Column.**

**Books.java**

1. **package** com.javatpoint.model;
2. **import** javax.persistence.Column;
3. **import** javax.persistence.Entity;
4. **import** javax.persistence.Id;
5. **import** javax.persistence.Table;
6. //mark class as an Entity
7. @Entity
8. //defining class name as Table name
9. @Table
10. **public** **class** Books
11. {
12. //Defining book id as primary key
13. @Id
14. @Column
15. **private** **int** bookid;
16. @Column
17. **private** String bookname;
18. @Column
19. **private** String author;
20. @Column
21. **private** **int** price;
22. **public** **int** getBookid()
23. {
24. **return** bookid;
25. }
26. **public** **void** setBookid(**int** bookid)
27. {
28. **this**.bookid = bookid;
29. }
30. **public** String getBookname()
31. {
32. **return** bookname;
33. }
34. **public** **void** setBookname(String bookname)
35. {
36. **this**.bookname = bookname;
37. }
38. **public** String getAuthor()
39. {
40. **return** author;
41. }
42. **public** **void** setAuthor(String author)
43. {
44. **this**.author = author;
45. }
46. **public** **int** getPrice()
47. {
48. **return** price;
49. }
50. **public** **void** setPrice(**int** price)
51. {
52. **this**.price = price;
53. }
54. }

**Step 11:** Create a package with the name **com.javatpoint.controller**in the folder **src/main/java.**

**Step 12:** Create a Controller class in the package **com.javatpoint.controller.**We have created a controller class with the name **BooksController.**In the BooksController class, we have done the following:

* Mark the class as **RestController**by using the annotation**@RestController.**
* Autowire the **BooksService** class by using the annotation **@Autowired**.
* Define the following methods:
  + **getAllBooks():** It returns a List of all Books.
  + **getBooks():** It returns a book detail that we have specified in the path variable. We have passed bookid as an argument by using the annotation @PathVariable. The annotation indicates that a method parameter should be bound to a URI template variable.
  + **deleteBook():** It deletes a specific book that we have specified in the path variable.
  + **saveBook():** It saves the book detail. The annotation @RequestBody indicates that a method parameter should be bound to the body of the web request.
  + **update():** The method updates a record. We must specify the record in the body, which we want to update. To achieve the same, we have used the annotation @RequestBody.

**BooksController.java**

1. **package** com.javatpoint.controller;
2. **import** java.util.List;
3. **import** org.springframework.beans.factory.annotation.Autowired;
4. **import** org.springframework.web.bind.annotation.DeleteMapping;
5. **import** org.springframework.web.bind.annotation.GetMapping;
6. **import** org.springframework.web.bind.annotation.PathVariable;
7. **import** org.springframework.web.bind.annotation.PostMapping;
8. **import** org.springframework.web.bind.annotation.PutMapping;
9. **import** org.springframework.web.bind.annotation.RequestBody;
10. **import** org.springframework.web.bind.annotation.RestController;
11. **import** com.javatpoint.model.Books;
12. **import** com.javatpoint.service.BooksService;
13. //mark class as Controller
14. @RestController
15. **public** **class** BooksController
16. {
17. //autowire the BooksService class
18. @Autowired
19. BooksService booksService;
20. //creating a get mapping that retrieves all the books detail from the database
21. @GetMapping("/book")
22. **private** List<Books> getAllBooks()
23. {
24. **return** booksService.getAllBooks();
25. }
26. //creating a get mapping that retrieves the detail of a specific book
27. @GetMapping("/book/{bookid}")
28. **private** Books getBooks(@PathVariable("bookid") **int** bookid)
29. {
30. **return** booksService.getBooksById(bookid);
31. }
32. //creating a delete mapping that deletes a specified book
33. @DeleteMapping("/book/{bookid}")
34. **private** **void** deleteBook(@PathVariable("bookid") **int** bookid)
35. {
36. booksService.delete(bookid);
37. }
38. //creating post mapping that post the book detail in the database
39. @PostMapping("/books")
40. **private** **int** saveBook(@RequestBody Books books)
41. {
42. booksService.saveOrUpdate(books);
43. **return** books.getBookid();
44. }
45. //creating put mapping that updates the book detail
46. @PutMapping("/books")
47. **private** Books update(@RequestBody Books books)
48. {
49. booksService.saveOrUpdate(books);
50. **return** books;
51. }
52. }

**Step 13:** Create a package with the name **com.javatpoint.service**in the folder **src/main/java.**

**Step 14:** Create a **Service** class. We have created a service class with the name **BooksService**in the package **com.javatpoint.service.**

**BooksService.java**

1. **package** com.javatpoint.service;
2. **import** java.util.ArrayList;
3. **import** java.util.List;
4. **import** org.springframework.beans.factory.annotation.Autowired;
5. **import** org.springframework.stereotype.Service;
6. **import** com.javatpoint.model.Books;
7. **import** com.javatpoint.repository.BooksRepository;
8. //defining the business logic
9. @Service
10. **public** **class** BooksService
11. {
12. @Autowired
13. BooksRepository booksRepository;
14. //getting all books record by using the method findaAll() of CrudRepository
15. **public** List<Books> getAllBooks()
16. {
17. List<Books> books = **new** ArrayList<Books>();
18. booksRepository.findAll().forEach(books1 -> books.add(books1));
19. **return** books;
20. }
21. //getting a specific record by using the method findById() of CrudRepository
22. **public** Books getBooksById(**int** id)
23. {
24. **return** booksRepository.findById(id).get();
25. }
26. //saving a specific record by using the method save() of CrudRepository
27. **public** **void** saveOrUpdate(Books books)
28. {
29. booksRepository.save(books);
30. }
31. //deleting a specific record by using the method deleteById() of CrudRepository
32. **public** **void** delete(**int** id)
33. {
34. booksRepository.deleteById(id);
35. }
36. //updating a record
37. **public** **void** update(Books books, **int** bookid)
38. {
39. booksRepository.save(books);
40. }
41. }

**Step 15:** Create a package with the name **com.javatpoint.repository**in the folder **src/main/java.**

**Step 16:** Create a **Repository** interface. We have created a repository interface with the name **BooksRepository**in the package **com.javatpoint.repository.**It extends the **Crud Repository** interface.

**BooksRepository.java**

1. **package** com.javatpoint.repository;
2. **import** org.springframework.data.repository.CrudRepository;
3. **import** com.javatpoint.model.Books;
4. //repository that extends CrudRepository
5. **public** **interface** BooksRepository **extends** CrudRepository<Books, Integer>
6. {
7. }

Now we will configure the datasource **URL, driver class name, username,**and**password,**in the **application.properties**file.

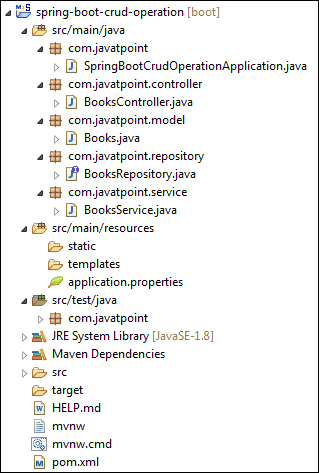
**Step 17:** Open the **application.properties** file and configure the following properties.

**application.properties**

1. spring.datasource.url=jdbc:h2:mem:books\_data
2. spring.datasource.driverClassName=org.h2.Driver
3. spring.datasource.username=sa
4. spring.datasource.password=
5. spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
6. #enabling the H2 console
7. spring.h2.console.enabled=**true**

Note: Do not forget to enable the H2 console.

After creating all the classes and packages, the project directory looks like the following.



Now we will run the application.

**Step 18:** Open **SpringBootCrudOperationApplication.java**file and run it as Java Application.

**SpringBootCrudOperationApplication.java**

1. **package** com.javatpoint;
2. **import** org.springframework.boot.SpringApplication;
3. **import** org.springframework.boot.autoconfigure.SpringBootApplication;
4. @SpringBootApplication
5. **public** **class** SpringBootCrudOperationApplication
6. {
7. **public** **static** **void** main(String[] args)
8. {
9. SpringApplication.run(SpringBootCrudOperationApplication.**class**, args);
10. }
11. }

Note: In the next steps we will use rest client Postman. So, ensure that the Postman application is already installed in your system.

**Step 19:** Open the **Postman**and do the following:

* Select the **POST**
* Invoke the URL http://localhost:8080/books.
* Select the **Body**
* Select he Content-Type **JSON (application/json).**
* Insert the data. We have inserted the following data in the Body:

1. {
2. "bookid": "5433",
3. "bookname": "Core and Advance Java",
4. "author": "R. Nageswara Rao",
5. "price": "800"
6. }

* Click on the **Send**

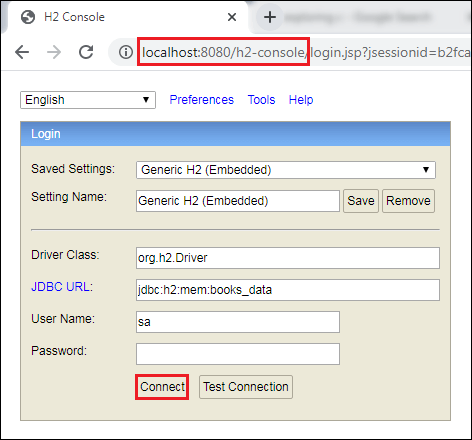
When the request is successfully executed, it shows the **Status:200 OK**. It means the record has been successfully inserted in the database.

Similarly, we have inserted the following data.

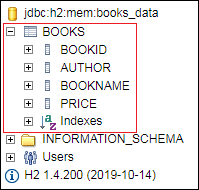
1. {
2. "bookid": "0982",
3. "bookname": "Programming with Java",
4. "author": "E. Balagurusamy",
5. "price": "350"
6. }
7. {
8. "bookid": "6321",
9. "bookname": "Data Structures and Algorithms in Java",
10. "author": "Robert Lafore",
11. "price": "590"
12. }
13. {
14. "bookid": "5433",
15. "bookname": "Effective Java",
16. "author": "Joshua Bloch",
17. "price": "670"
18. }

Let's access the H2 console to see the data.

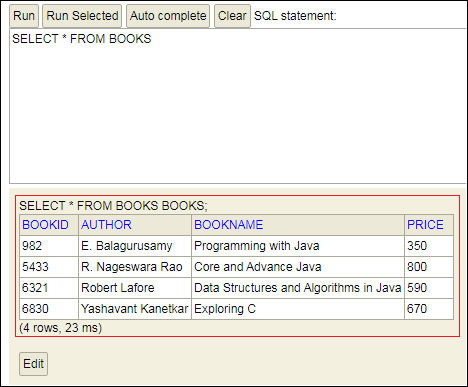
**Step 20:** Open the browser and invoke the URL http://localhost:8080/h2-console. Click on the **Connect** button, as shown below.



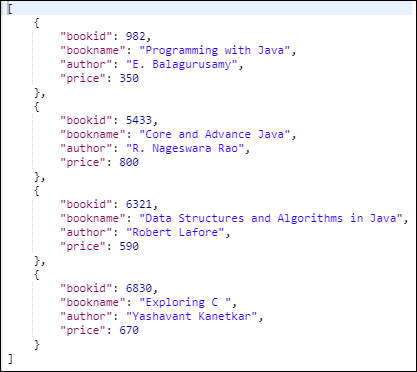
After clicking on the **Connect** button, we see the **Books** table in the database, as shown below.



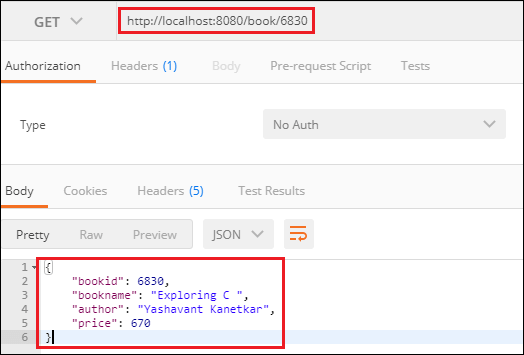
**Step 21:** Click on the **Books** table and then click on the **Run** button. The table shows the data that we have inserted in the body.



**Step 22:** Open the **Postman** and send a **GET** request with the URL http://localhost:8080/books. It returns the data that we have inserted in the database.

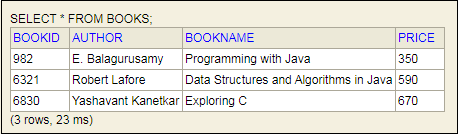


Let's send a **GET** request with the URL http://localhost:8080/book/{bookid}. We have specified the **bookid 6830**. It returns the detail of the book whose id is 6830.



Similarly, we can also send a **DELETE** request to delete a record. Suppose we want to delete a book record whose id is **5433**.

Select the **DELETE** method and invoke the URL http://localhost:8080/book/5433. Again, execute the **Select** query in the H2 console. We see that the book whose id is **5433** has been deleted from the database



Similarly, we can also update a record by sending a **PUT** request. Let's update the price of the book whose id is **6321**.

* Select the **PUT**
* In the request body, paste the record which you want to update and make the changes. In our case, we want to update the record of the book whose id is 6321. In the following record, we have changed the price of the book.

1. {
2. "bookid": "6321",
3. "bookname": "Data Structures and Algorithms in Java",
4. "author": "Robert Lafore",
5. "price": "500"
6. }

* Click on the **Send**

Now, move to the H2 console and see the changes have reflected or not. We see that the price of the book has been changed, as shown below.

