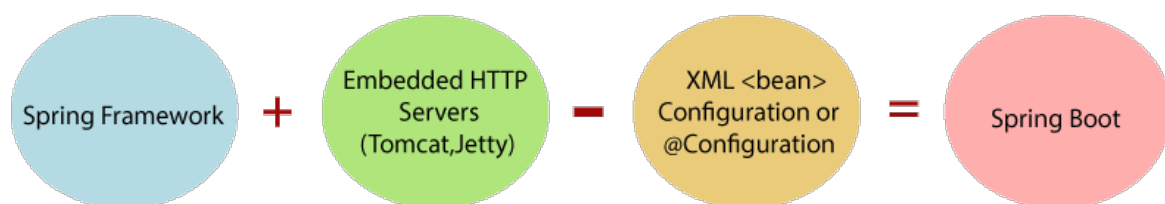


SPRINGBOOT

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 =8082** 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 vs. Spring Boot

Spring: Spring Framework is the most popular application development framework of Java. The main feature of the Spring Framework is **dependency Injection** or **Inversion of Control** (IoC). With the help of Spring Framework, we can develop a **loosely** coupled application. It is better to use if application type or characteristics are purely defined.

Spring Boot: Spring Boot is a module of Spring Framework. It allows us to build a stand-alone application with minimal or zero configurations. It is better to use if we want to develop a simple Spring-based application or RESTful services.

The primary comparison between Spring and Spring Boot are discussed below:

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 **stand-alone** applications with less **loosely coupled** 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 server 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 vs. Spring MVC

Spring Boot: Spring Boot makes it easy to quickly bootstrap and start developing a Spring-based application. It avoids a lot of boilerplate code. It hides a lot of complexity behind the

scene so that the developer can quickly get started and develop Spring-based applications easily.

Spring MVC: Spring MVC is a Web MVC Framework for building web applications. It contains a lot of configuration files for various capabilities. It is an HTTP oriented web application development framework.

Spring Boot and Spring MVC exist for different purposes. The primary comparison between Spring Boot and Spring MVC are discussed below:

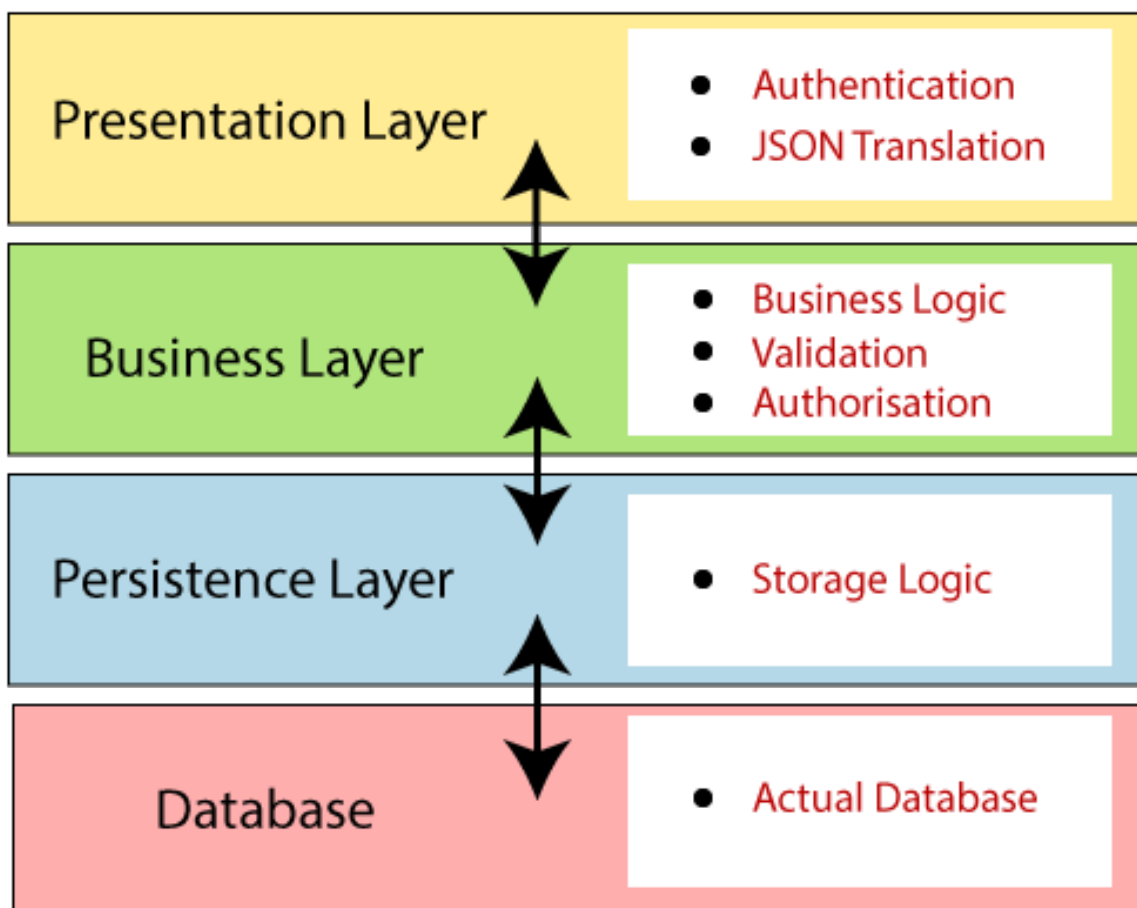
Spring Boot	Spring MVC
Spring Boot is a module of Spring for packaging the Spring-based application with sensible defaults. It provides default configurations to build Spring-powered framework. There is no need to build configuration manually. There is no requirement for a deployment descriptor. It avoids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

- **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.

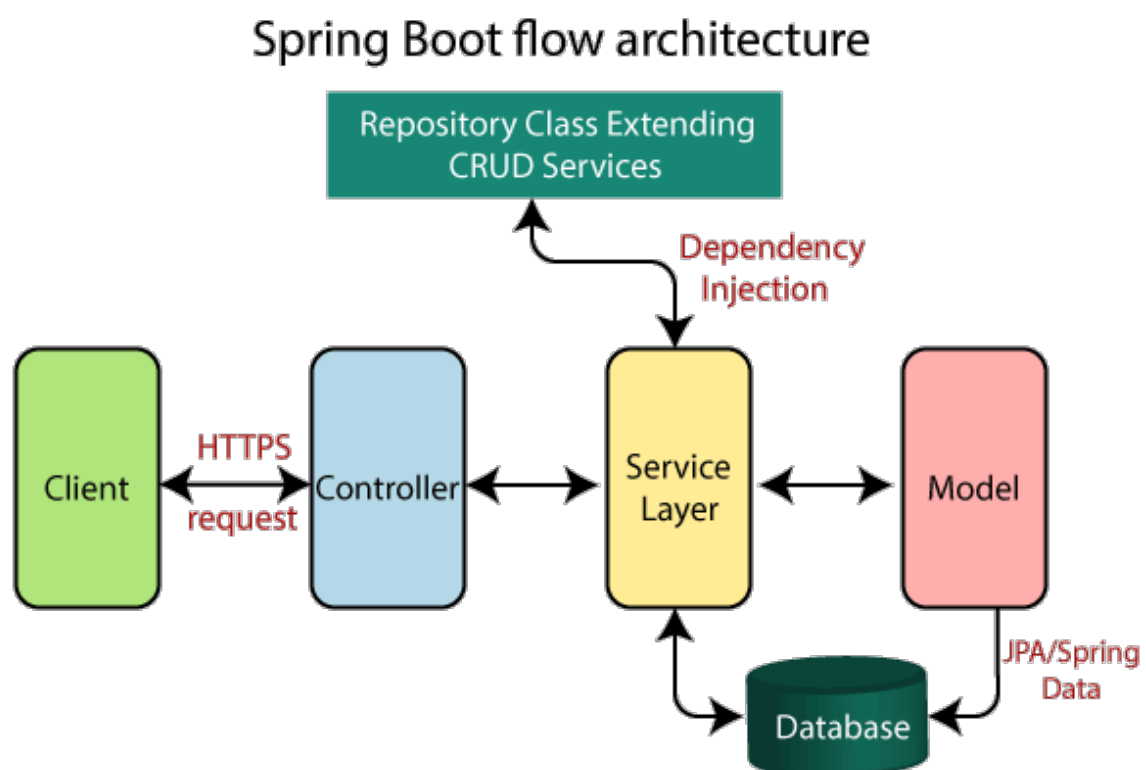
In short, it consists of **views** i.e., frontend part.

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.

Spring Initializr

Spring Initializr is a **web-based tool** provided by the Pivotal Web Service. With the help of **Spring Initializr**, we can easily generate the structure of the **Spring Boot Project**. It offers extensible API for creating JVM-based projects.

It also provides various options for the project that are expressed in a metadata model. The metadata model allows us to configure the list of dependencies supported by JVM and platform versions, etc. It serves its metadata in a well-known that provides necessary assistance to third-party clients.

Spring Initializr Modules

Spring Initializr has the following module:

- **initializr-actuator:** It provides additional information and statistics on project generation. It is an optional module.
- **initializr-bom:** In this module, **BOM** stands for **Bill Of**

Materials. In Spring Boot, BOM is a special kind of **POM** that is used to control the **versions** of a project's **dependencies**. It provides a central place to define and update those versions. It provides flexibility to add a dependency in our module without worrying about the versions.

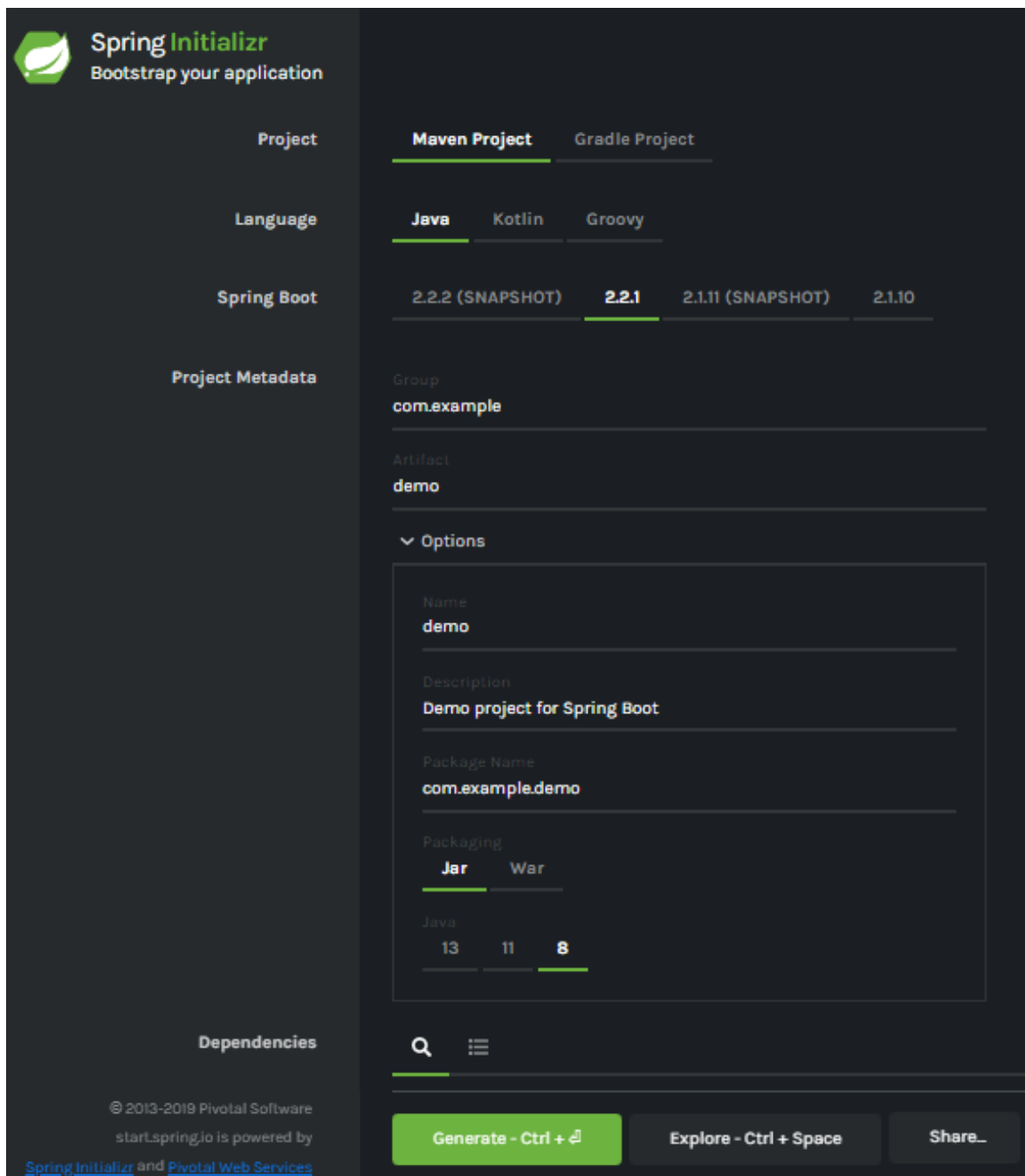
Outside the software world, the **BOM** is a list of parts, items, assemblies, and other materials required to create products. It explains **what**, **how**, and **where** to collect required materials.

- **initializr-docs:** It provides documentation.
- **initializr-generator:** It is a core project generation library.
- **initializr-generator-spring:**
- **initializr-generator-test:** It provides a test infrastructure for project generation.
- **initializr-metadata:** It provides metadata infrastructure for various aspects of the projects.
- **initializr-service-example:** It provides custom instances.
- **initializr-version-resolver:** It is an optional module to extract version numbers from an arbitrary POM.
- **initializr-web:** It provides web endpoints for third party clients.

Supported Interface

- It supports **IDE STS, IntelliJ IDEA Ultimate, NetBeans, Eclipse**. You can download the plugin from <https://github.com/AlexFalappa/nb-springboot>. If you are using VSCode, download the plugin from <https://github.com/microsoft/vscode-spring-initializr>.
- Use Custom Web UI <http://start.spring.io> or <https://start-scs.cfapps.io>.
- It also supports the command-line with the **Spring Boot CLI** or **cURL** or **HTTPIe**.

The following image shows the Spring Initializr UI:

A screenshot of the Spring Initializr web application. The interface is dark-themed. On the left is a sidebar with navigation links: Project, Language, Spring Boot, Project Metadata, and Dependencies. The main area shows configuration options: Maven Project (selected), Java (selected), 2.2.1 (selected), Group (com.example), Artifact (demo), and a collapsed Options section. The Options section contains fields for Name (demo), Description (Demo project for Spring Boot), Package Name (com.example.demo), Packaging (Jar selected), and Java version (8 selected). At the bottom are buttons for Generate, Explore, and Share.

Spring Initializr
Bootstrap your application

Project **Maven Project** Gradle Project

Language **Java** Kotlin Groovy

Spring Boot 2.2.2 (SNAPSHOT) **2.2.1** 2.1.11 (SNAPSHOT) 2.1.10

Project Metadata

Group
com.example

Artifact
demo

Options

Name
demo

Description
Demo project for Spring Boot

Package Name
com.example.demo

Packaging
Jar War

Java
13 11 **8**

Dependencies

© 2013-2019 Pivotal Software
start.spring.io is powered by
[Spring Initializr](#) and [Pivotal Web Services](#)

Generate - Ctrl + G **Explore - Ctrl + Space** **Share...**

Generating a Project

Before creating a project, we must be friendly with UI. Spring Initializr UI has the following labels:

- **Project:** It defines the **kind** of project. We can create either **Maven Project** or **Gradle Project**. We will create a **Maven Project** throughout the tutorial.

- **Language:** Spring Initializr provides the choice among three languages **Java**, **Kotlin**, and **Groovy**. Java is by default selected.
- **Spring Boot:** We can select the Spring Boot **version**. The latest version is **2.2.2**.
- **Project Metadata:** It contains information related to the project, such as **Group**, **Artifact**, etc. **Group** denotes the **package** name; **Artifact** denotes the **Application** name. The default **Group** name is **com.example**, and the default **Artifact** name is **demo**.
- **Dependencies:** Dependencies are the collection of artifacts that we can add to our project.

There is another **Options** section that contains the following fields:

- **Name:** It is the same as **Artifact**.
- **Description:** In the description field, we can write a **description** of the project.
- **Package Name:** It is also similar to the **Group** name.
- **Packaging:** We can select the **packing** of the project. We can choose either **Jar** or **War**.
- **Java:** We can select the **JVM** version which we want to use. We will use **Java 8** version throughout the tutorial.

There is a **Generate** button. When we click on the button, it starts packing the project and downloads the **Jar** or **War** file, which you have selected.

Download and Install STS IDE

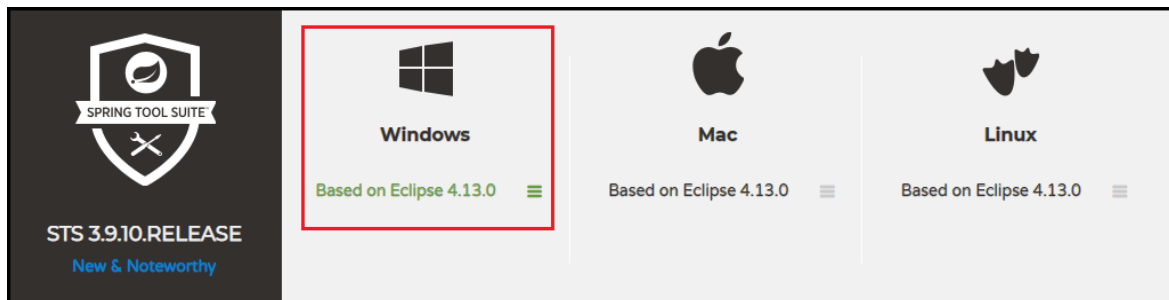
Spring Tool Suite (STS) IDE

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.

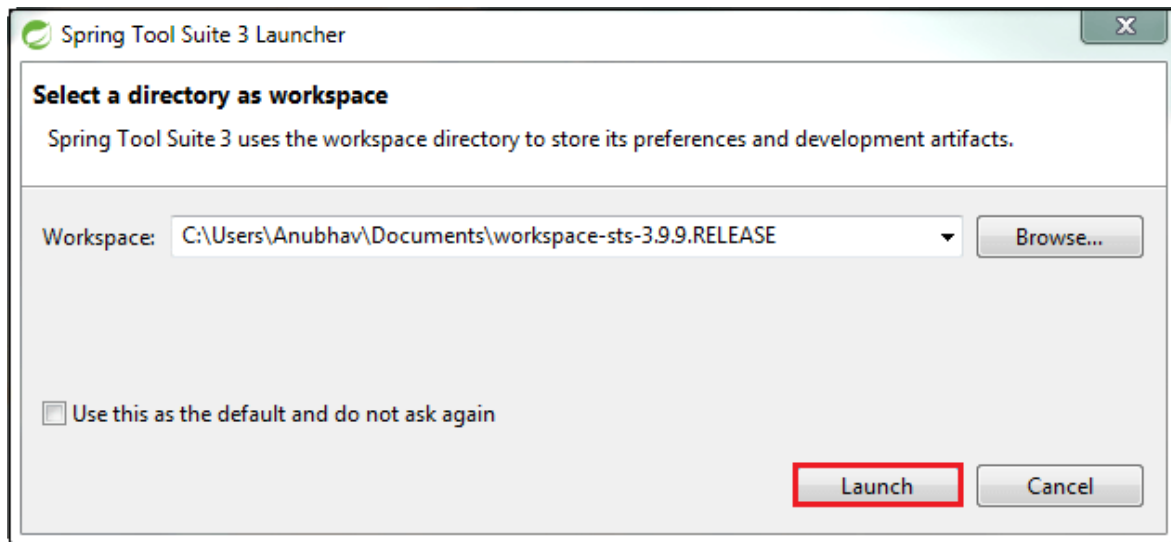


Step 2: Extract the **zip** file and install the STS.

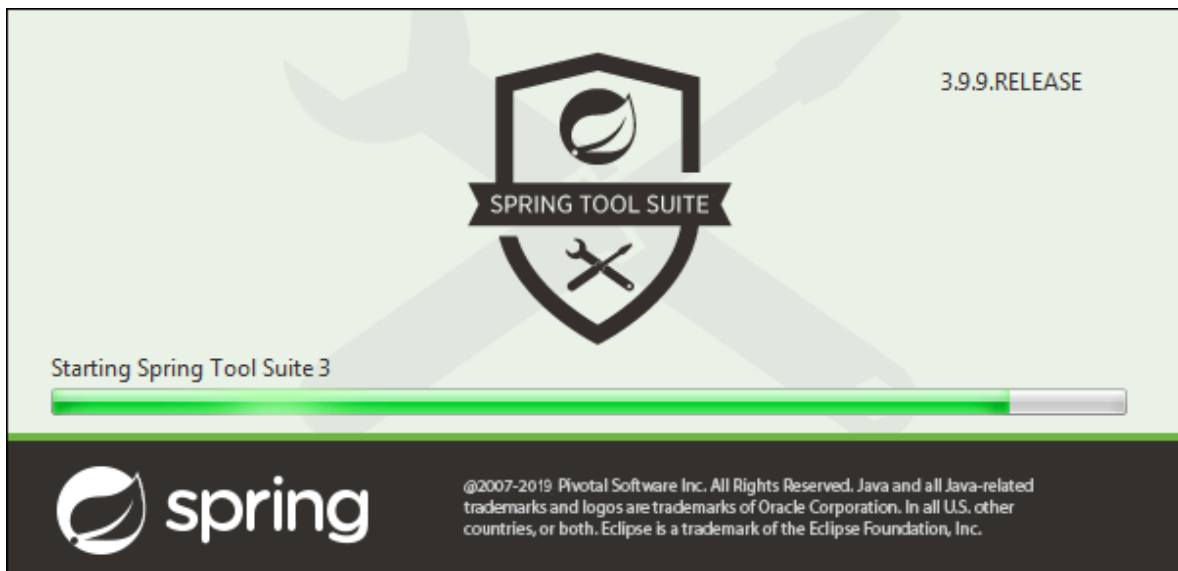
sts-bundle -> sts-3.9.9.RELEASE -> Double-click on the **STS.exe**.

license.txt	11,522	4,167	Text Document
open-source-licenses.txt	1,619,852	64,513	Text Document
STS.exe	417,280	74,922	Application
STS.ini	394	251	Configuration setti...

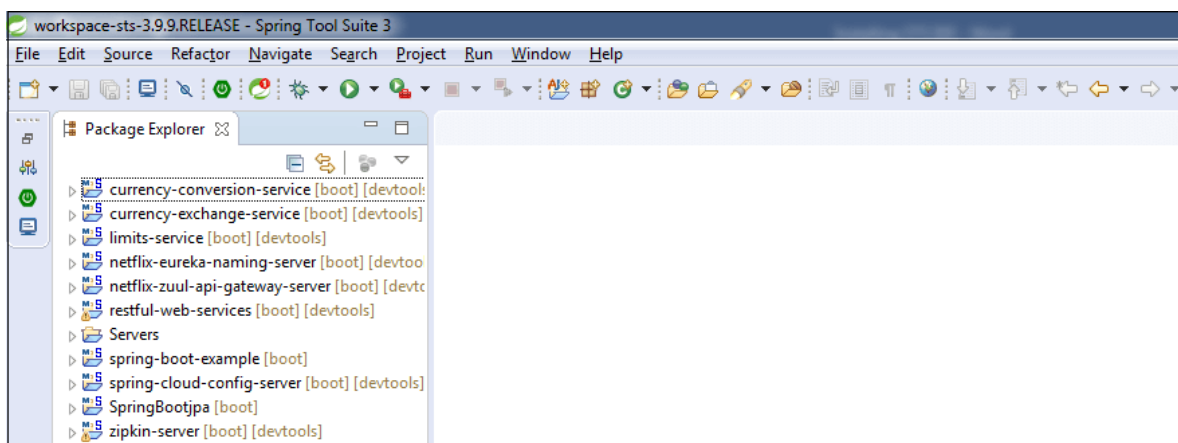
Step 3: Spring Tool Suite 3 Launcher dialog box appears on the screen. Click on the **Launch** button. You can change the Workspace if you want.



Step 4: It starts launching the STS.



The STS user interface looks like the following:



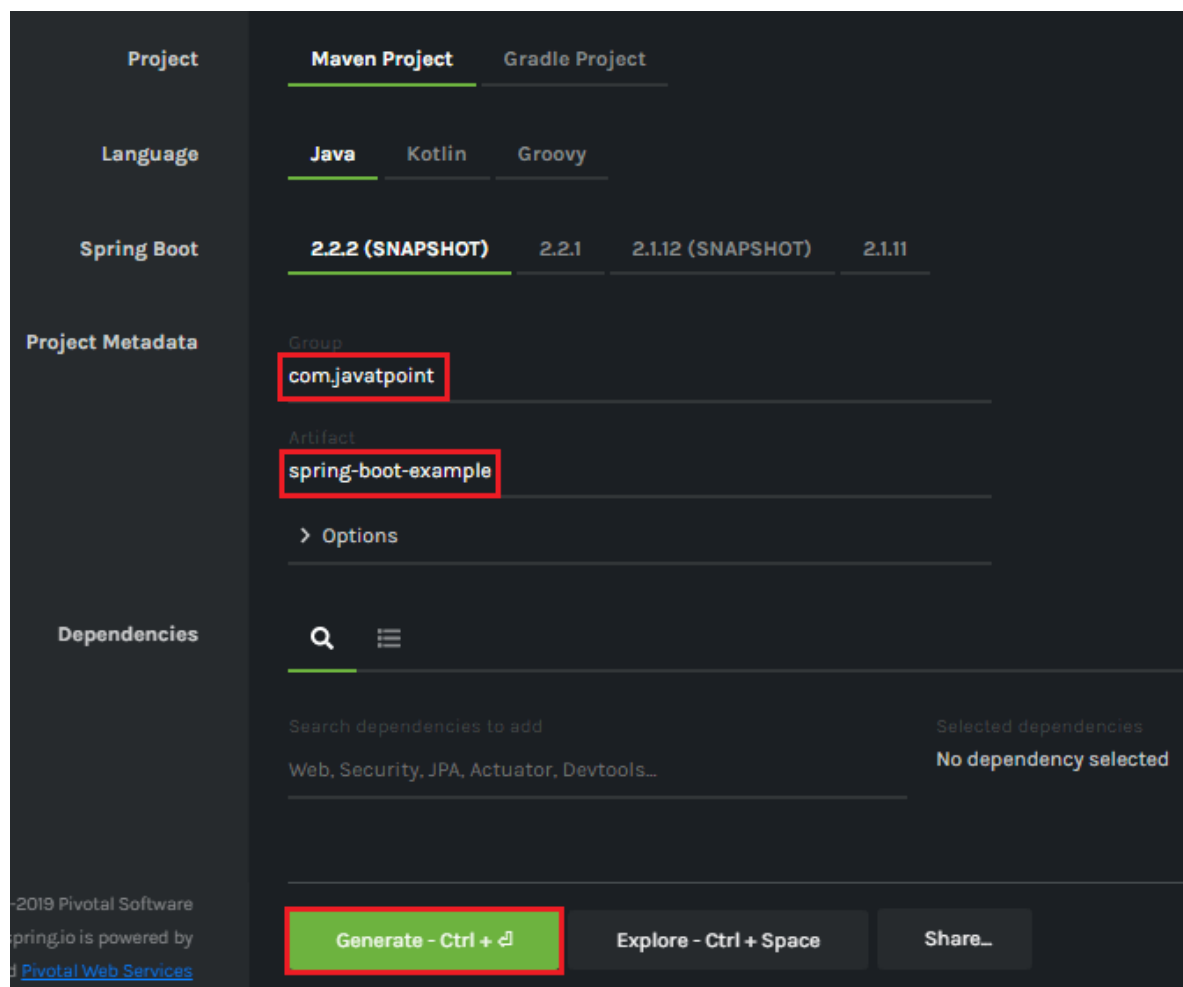
Creating a Spring Boot Project

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

Step 1: Open the Spring initializer <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.



The screenshot displays the Spring Initializr web application interface. On the left is a dark sidebar with navigation links: Project, Language, Spring Boot, Project Metadata, and Dependencies. The main content area is light gray and shows the configuration for a Maven Project. Under 'Language', 'Java' is selected. Under 'Spring Boot', '2.2.2 (SNAPSHOT)' is selected. In the 'Project Metadata' section, the 'Group' field contains 'com.javatpoint' and the 'Artifact' field contains 'spring-boot-example', both highlighted with red boxes. Below these fields is an 'Options' section with a right-pointing arrow. The 'Dependencies' section features a search bar with a magnifying glass icon and a list of suggested dependencies: 'Web, Security, JPA, Actuator, Devtools...'. To the right of the search bar, it says 'Selected dependencies: No dependency selected'. At the bottom of the page, there are three buttons: 'Generate - Ctrl + G' (highlighted with a red box), 'Explore - Ctrl + Space', and 'Share...'. The footer text at the bottom left reads: '© 2019 Pivotal Software, Inc. Spring.io is powered by Pivotal Web Services'.

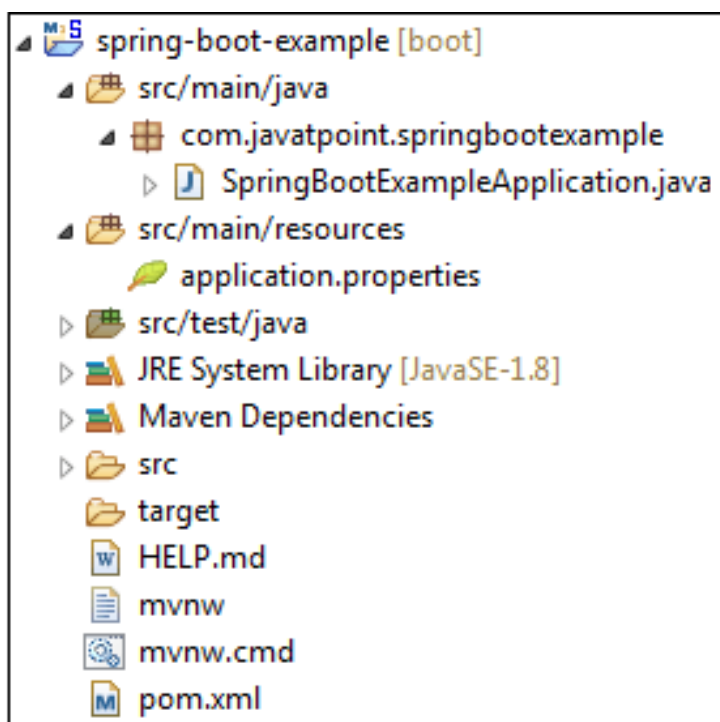
When we click on the Generate button, it starts packing the project in a **.rar** file and downloads the project.

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  
i       m       p       o       r       t
```

```

    org.springframework.boot.autoconfigure.SpringBootApplication;
4  @SpringBootApplication
5  public class SpringBootApplicationExampleApplication
6  {
7  public static void main(String[] args)
8  {
9
    SpringApplication.run(SpringBootApplicationExampleApplication.class, args);
10 }
11 }

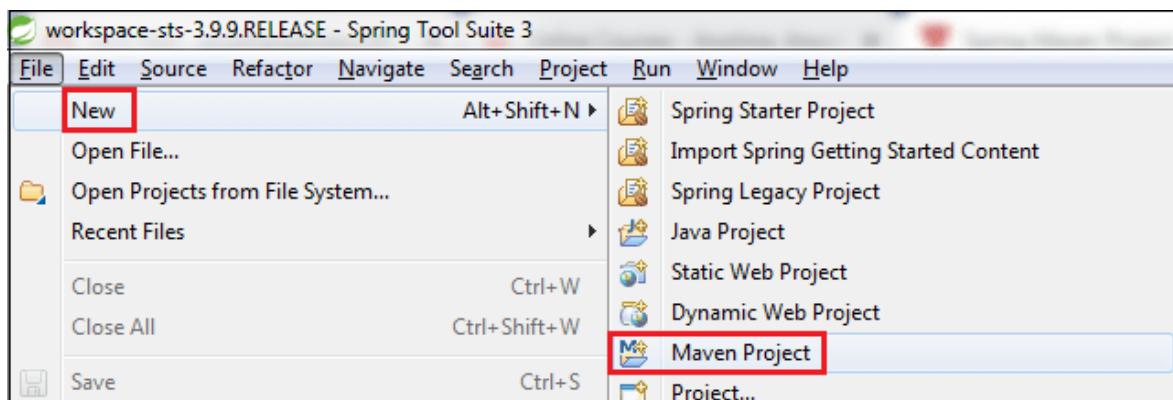
```

Creating a Spring Boot Project Using STS

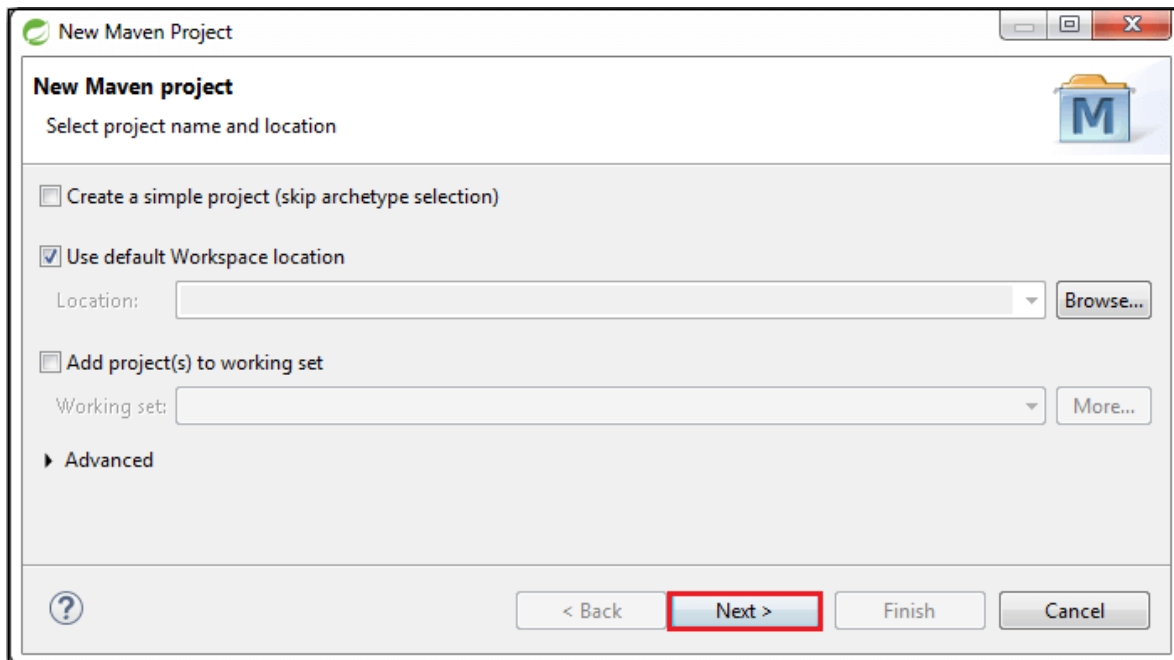
We can also use Spring Tool Suite to create a Spring project. In this section, we will create a **Maven Project** using **STS**.

Step 1: Open the Spring Tool Suite.

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

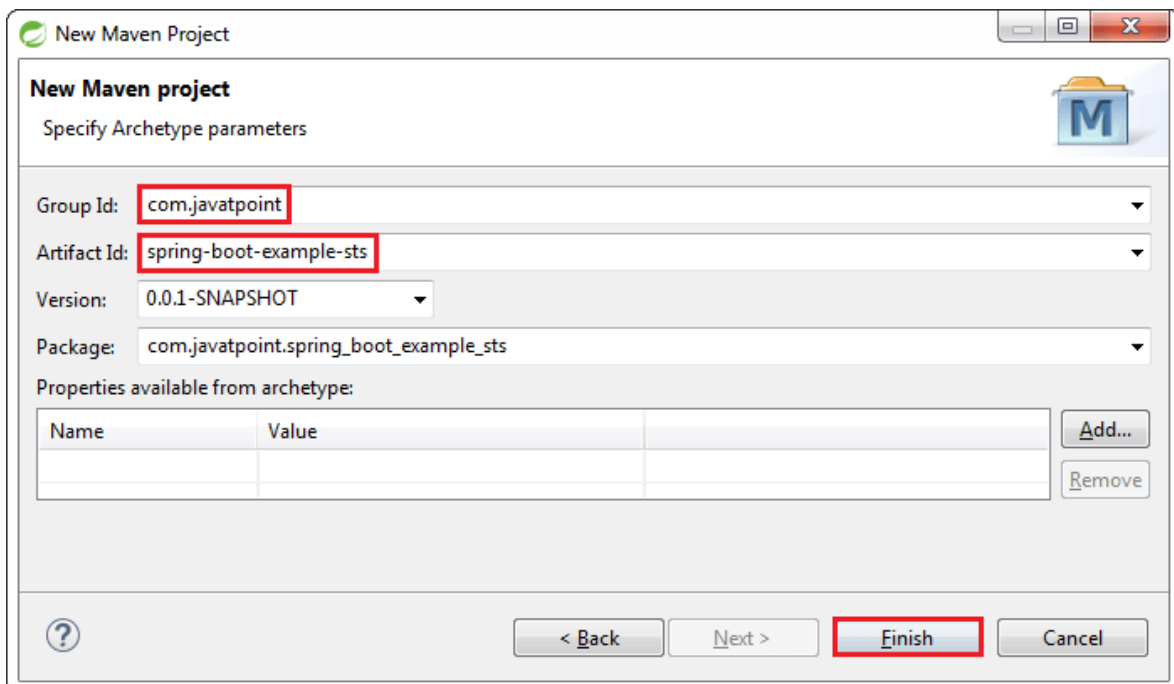


It shows the New Maven Project wizard. Click on the **Next** button.



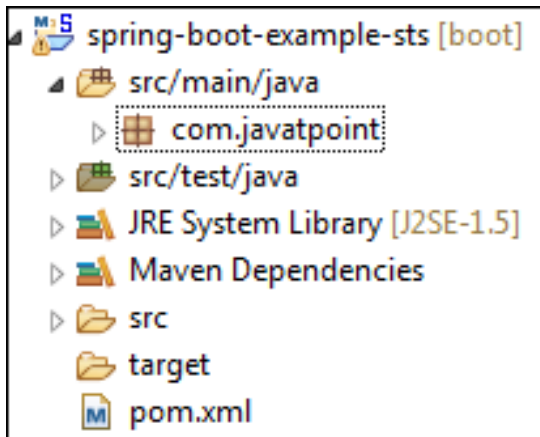
Step 3: Select the **maven-archetype-quickstart** and click on the **Next** button.

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.

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 **BeanInitializationException**.

```
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 = "/"
6         {name}", method = RequestMethod.GET)
7     public Employee getBooksByName()
8     {
9         return booksTemplate;
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     {

```

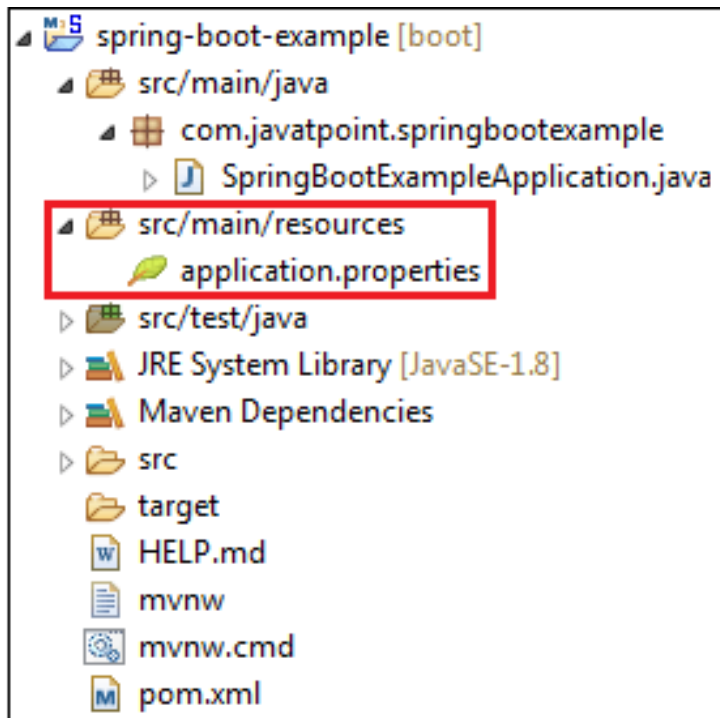
```
7 //application code
8 return "bookList";
9 }
```

- **@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 from 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 times 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
```

In the above example, we have configured the **application**

name and **port**. The port 8081 denotes that the application runs on port **8081**.

YAML Properties File

Spring Boot provides another file to configure the properties is called **yml** file. The Yaml file works because the **Snake YAML** jar is present in the classpath. Instead of using the application.properties file, we can also use the application.yml file, but the **Yml** file should be present in the classpath.

Example of application.yml

```
1  spring:
2  application:
3  name: demoApplication
4  server:
5  port: 8081
```

In the above example, we have configured the **application name** and **port**. The port 8081 denotes that the application runs on port **8081**.

Spring Boot Property Categories

There are **sixteen** categories of Spring Boot Property are as follows:

- 1 Core Properties
- 2 Cache Properties
- 3 Mail Properties
- 4 JSON Properties
- 5 Data Properties
- 6 Transaction Properties
- 7 Data Migration Properties
- 8 Integration Properties
- 9 Web Properties
- 10 Templating Properties

- 11 Server Properties
- 12 Security Properties
- 13 RSocket Properties
- 14 Actuator Properties
- 15 DevTools Properties
- 16 Testing Properties

Spring Boot Starters

Spring Boot provides a number of **starters** that allow us to add jars in the classpath. Spring Boot built-in **starters** make development easier and rapid. **Spring Boot Starters** are the **dependency descriptors**.

In the Spring Boot Framework, all the starters follow a similar naming pattern: **spring-boot-starter-***, where ***** denotes a particular type of application. For example, if we want to use Spring and JPA for database access, we need to include the **spring-boot-starter-data-jpa** dependency in our **pom.xml** file of the project.

Third-Party Starters

We can also include **third party starters** in our project. But we do not use **spring-boot-starter** for including third party dependency. The **spring-boot-starter** is reserved for official Spring Boot artifacts. The third-party starter starts with the name of the project. For example, the third-party project name is **abc**, then the dependency name will be **abc-spring-boot-starter**.

The Spring Boot Framework provides the following application starters under the **org.springframework.boot** group.

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 Starter Parent

Spring Boot Starter Parent

The spring-boot-starter-parent is a project starter. It provides default configurations for our applications. It is used internally by all dependencies. All Spring Boot projects use spring-boot-starter-parent as a parent in pom.xml file.

```

1 <parent>
2 <groupId>org.springframework.boot</groupId>
3 <artifactId>spring-boot-starter-parent</artifactId>
4 <version>1.4.0.RELEASE</version>
5 </parent>
```

Parent Poms allow us to manage the following things for multiple child projects and modules:

- **Configuration:** It allows us to maintain consistency of Java Version and other related properties.
- **Dependency Management:** It controls the versions of dependencies to avoid conflict.
- Source encoding
- Default Java Version
- Resource filtering
- It also controls the default plugin configuration.

The spring-boot-starter-parent inherits dependency

management from spring-boot-dependencies. We only need to specify the Spring Boot version number. If there is a requirement of the additional starter, we can safely omit the version number.

Spring Boot Starter Parent Internal

Spring Boot Starter Parent defines spring-boot-dependencies as a parent pom. It inherits dependency management from spring-boot-dependencies.

```
1 <parent>
2 <groupId>org.springframework.boot</groupId>
3 <artifactId>spring-boot-dependencies</artifactId>
4 <version>1.6.0.RELEASE</version>
5 <relativePath>../../spring-boot-dependencies</
  relativePath>
6 </parent>
```

Default Parent Pom

```
1 <properties>
2 <java.version>1.8</java.version>
3 <resource.delimiter>@</resource.delimiter>
4 <project.build.sourceEncoding>UTF-8</
  project.build.sourceEncoding>
5 <project.reporting.outputEncoding>UTF-8</
  project.reporting.outputEncoding>
6 <maven.compiler.source>${java.version}</
  maven.compiler.source>
7 <maven.compiler.target>${java.version}</
  maven.compiler.target>
8 </properties>
```

The properties section defines the application default values. The default Java version is 1.8. We can also override Java

version by specifying a property `<java.version>1.8</java.version>` in the project pom. The parent pom also contains the few other settings related to encoding and source. The Spring Boot framework uses these defaults in case, if we have not defined in the application.properties file.

Plugin Management

The **spring-boot-starter-parent** specifies the default configuration for a host of plugins including maven-failsafe-plugin, maven-jar-plugin and maven-surefire-plugin.

```
1  <plugin>
2  <groupId>org.apache.maven.plugins</groupId>
3  <artifactId>maven-failsafe-plugin</artifactId>
4  <executions>
5  <execution>
6  <goals>
7  <goal>integration-test</goal>
8  <goal>verify</goal>
9  </goals>
10 </execution>
11 </executions>
12 </plugin>
13 <plugin>
14 <groupId>org.apache.maven.plugins</groupId>
15 <artifactId>maven-jar-plugin</artifactId>
16 <configuration>
17 <archive>
18 <manifest>
19 <mainClass>${start-class}</mainClass> <addDefaultImplementationEntries>true</addDefaultImplementationEntries>
20 </manifest>
21 </archive>
22 </configuration>
23 </plugin>
```

```

24 <plugin>
25 <groupId>org.apache.maven.plugins</groupId>
26 <artifactId>maven-surefire-plugin</artifactId>
27 <configuration>
28 <includes>
29 <include>**/*Tests.java</include>
30 <include>**/*Test.java</include>
31 </includes>
32 <excludes>
33 <exclude>**/Abstract*.java</exclude>
34 </excludes>
35 </configuration>
36 </plugin>

```

Spring Boot Dependencies

The spring-boot-starter-parent dependency inherit from the spring-boot-dependencies, it shares all these characteristics as well. Hence the Spring Boot manages the list of the dependencies as the part of the dependency management.

```

1 <properties>
2 <activemq.version>5.13.4</activemq.version>
3 ...
4 <ehcache.version>2.10.2.2.21</ehcache.version>
5 <ehcache3.version>3.1.1</ehcache3.version>
6 ...
7 <h2.version>1.4.192</h2.version>
8 <hamcrest.version>1.3</hamcrest.version>
9 <hazelcast.version>3.6.4</hazelcast.version>
10 <hibernate.version>5.0.9.Final</hibernate.version>
11 <hibernate-validator.version>5.2.4.Final</hibernate-
    validator.version>
12 <hikaricp.version>2.4.7</hikaricp.version>
13 <hikaricp-java6.version>2.3.13</hikaricp-java6.version>

```

```
14 <hornetq.version>2.4.7.Final</hornetq.version>
15 <hsqldb.version>2.3.3</hsqldb.version>
16 <htmlunit.version>2.21</htmlunit.version>
17 <httpasyncclient.version>4.1.2</
  httpasyncclient.version>
18 <httpclient.version>4.5.2</httpclient.version>
19 <httpcore.version>4.4.5</httpcore.version>
20 <infinispan.version>8.2.2.Final</infinispan.version>
21 <jackson.version>2.8.1</jackson.version>
22 ...
23 <jersey.version>2.23.1</jersey.version>
24 <jest.version>2.0.3</jest.version>
25 <jetty.version>9.3.11.v20160721</jetty.version>
26 <jetty-jsp.version>2.2.0.v201112011158</jetty-
  jsp.version>
27 <spring-security.version>4.1.1.RELEASE</spring-
  security.version>
28 <tomcat.version>8.5.4</tomcat.version>
29 <undertow.version>1.3.23.Final</undertow.version>
30 <velocity.version>1.7</velocity.version>
31 <velocity-tools.version>2.0</velocity-tools.version>
32 <webjars-hal-browser.version>9f96c74</webjars-hal-
  browser.version>
33 <webjars-locator.version>0.32</webjars-
  locator.version>
34 <wsdl4j.version>1.6.3</wsdl4j.version>
35 <xml-apis.version>1.4.01</xml-apis.version>
36 </properties>
37 <prerequisites>
38 <maven>3.2.1</maven>
39 </prerequisites>
```

Spring Boot Starter without Parent

In some cases, we need not to inherit spring-boot-starter-parent in the pom.xml file. To handle such use cases, Spring Boot provides the flexibility to still use the dependency management without inheriting the spring-boot-starter-parent.


```
1 <dependencyManagement>
2 <dependencies>
3 <dependency>
4 <!-- Import dependency management from Spring Boot --
5 <groupId>org.springframework.boot</groupId>
6 <artifactId>spring-boot-dependencies</artifactId>
7 <version>2.1.1.RELEASE</version>
8 <type>pom</type>
9 <scope>import</scope>
10 </dependency>
11 </dependencies>
12 </dependencyManagement>
```

In the above code, we can see that we have used **<scope>** tag for this. It is useful when we want to use different version for a certain dependency.

Spring Boot Starter Web

There are two important features of spring-boot-starter-web:

- It is compatible for web development
- Auto configuration

If we want to develop a web application, we need to add the following dependency in pom.xml file:

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

Starter of Spring web uses Spring MVC, REST and Tomcat as a default embedded server. The single spring-boot-starter-web dependency transitively pulls in all dependencies related to web development. It also reduces the build dependency count.

The spring-boot-starter-web transitively depends on the following:

- org.springframework.boot:spring-boot-starter
- org.springframework.boot:spring-boot-starter-tomcat
- org.springframework.boot:spring-boot-starter-validation
- com.fasterxml.jackson.core:jackson-databind
- org.springframework:spring-web
- org.springframework:spring-webmvc

By default, the spring-boot-starter-web contains the following tomcat server dependency:

```
1 <dependency>
2 <groupId>org.springframework.boot</groupId>
3 <artifactId>spring-boot-starter-tomcat</artifactId>
4 <version>2.0.0.RELEASE</version>
5 <scope>compile</scope>
6 </dependency>
```

The spring-boot-starter-web auto-configures the following things that are required for the web development:

- Dispatcher Servlet
- Error Page
- Web JARs for managing the static dependencies
- Embedded servlet container

Spring Boot Embedded Web Server

Each Spring Boot application includes an embedded server. Embedded server is embedded as a part of deployable application. The advantage of embedded server is, we do not require pre-installed server in the environment. With Spring Boot, default embedded server is **Tomcat**. Spring Boot also supports another two embedded servers:

- **Jetty Server**
- **Undertow Server**

Using another embedded web server

For **servlet stack** applications, the **spring-boot-starter-web** includes **Tomcat** by including **spring-boot-starter-tomcat**, but we can use **spring-boot-starter-jetty** or **spring-boot-starter-undertow** instead.

For **reactive stack** applications, the **spring-boot-starter-webflux** includes **Reactor Netty** by including **spring-boot-starter-reactor-netty**, but we can use **spring-boot-starter-tomcat**, **spring-boot-starter-jetty**, or **spring-boot-starter-undertow** instead.

Jetty Server

The Spring Boot also supports an embedded server called **Jetty Server**. It is an HTTP server and Servlet container that has the capability of serving static and dynamic content. It is used when machine to machine communication is required.

If we want to add the Jetty server in the application, we need to add the **spring-boot-starter-jetty** dependency in our pom.xml file.

Remember: While using Jetty server in the application, make sure that the default Tomcat server is **excluded** from the **spring-boot-starter-web**. It avoids the conflict between servers.

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

```
4 <exclusions>
5 <exclusion>
6 <groupId>org.springframework.boot</groupId>
7 <artifactId>spring-boot-starter-tomcat</artifactId>
8 </exclusion>
9 </exclusions>
10 </dependency>
11 <dependency>
12 <groupId>org.springframework.boot</groupId>
13 <artifactId>spring-boot-starter-jetty</artifactId>
14 </dependency>
```

We can also customize the behavior of the Jetty server by using the **application.properties** file.

Undertow Server

Spring Boot provides another server called **Undertow**. It is also an embedded web server like Jetty. It is written in Java and managed and sponsored by JBoss. The main advantages of Undertow server are:

- Supports HTTP/2
- HTTP upgrade support
- Websocket Support
- Provides support for Servlet 4.0
- Flexible
- Embeddable

Remember: While using Undertow server in the application, make sure that the default Tomcat server is **excluded** from the **spring-boot-starter-web**. It avoids the conflict between servers.

```
1 <dependency>
2 <groupId>org.springframework.boot</groupId>
3 <artifactId>spring-boot-starter-web</artifactId>
4 <exclusions>
```

```
5 <exclusion>
6 <groupId>org.springframework.boot</groupId>
7 <artifactId>spring-boot-starter-tomcat</artifactId>
8 </exclusion>
9 </exclusions>
10 </dependency>
11 <dependency>
12 <groupId>org.springframework.boot</groupId>
13 <artifactId>spring-boot-starter-undertow</artifactId>
14 </dependency>
```

We can also customize the behavior of the Undertow server by using the **application.properties** file.

spring-boot-starter-web vs. spring-boot-starter-tomcat

The spring-boot-starter-web contains the spring web dependencies that includes spring-boot-starter-tomcat. The spring-boot-starter-web contains the following:

- spring-boot-starter
- jackson
- spring-core
- spring-mvc
- spring-boot-starter-tomcat

While the **spring-boot-starter-tomcat** contains everything related to Tomcat server.

- core
- el
- logging
- websocket

The starter-tomcat has the following dependencies:

```
1 <dependency>
2 <groupId>org.apache.tomcat.embed</groupId>
```

```
3 <artifactId>tomcat-embed-core</artifactId>
4 <version>8.5.23</version>
5 <scope>compile</scope>
6 </dependency>
7 <dependency>
8 <groupId>org.apache.tomcat.embed</groupId>
9 <artifactId>tomcat-embed-el</artifactId>
10 <version>8.5.23</version>
11 <scope>compile</scope>
12 </dependency>
13 <dependency>
14 <groupId>org.apache.tomcat.embed</groupId>
15 <artifactId>tomcat-embed-websocket</artifactId>
16 <version>8.5.23</version>
17 <scope>compile</scope>
18 </dependency>
```

We can also use **spring-mvc** without using the embedded Tomcat server. If we want to do so, we need to exclude the Tomcat server by using the **<exclusion>** tag, as shown in the following code.

```
1 <dependency>
2 <groupId>org.springframework.boot</groupId>
3 <artifactId>spring-boot-starter-web</artifactId>
4 <exclusions>
5 <exclusion>
6 <groupId>org.springframework.boot</groupId>
7 <artifactId>spring-boot-starter-tomcat</artifactId>
8 </exclusion>
9 </exclusions>
10 </dependency>
```

Spring Data JPA

Spring Data is a high-level Spring Source project. Its purpose is to unify and easy access to the different kinds of persistence

stores, both relational database systems, and NoSQL data stores.

When we implement a new application, we should focus on the business logic instead of technical complexity and boilerplate code. That's why the Java Persistent API (JPA) specification and Spring Data JPA are extremely popular.

Spring Data JPA adds a layer on the top of JPA. It means, Spring Data JPA uses all features defined by JPA specification, especially the entity, association mappings, and JPA's query capabilities. Spring Data JPA adds its own features such as the no-code implementation of the repository pattern and the creation of database queries from the method name.

Spring Data JPA

Spring Data JPA handles most of the complexity of JDBC-based database access and ORM (Object Relational Mapping). It reduces the boilerplate code required by JPA. It makes the implementation of your persistence layer easier and faster.

Spring Data JPA aims to improve the implementation of data access layers by reducing the effort to the amount that is needed.

Spring Data JPA Features

There are **three** main features of Spring Data JPA are as follows:

- **No-code repository:** It is the most popular persistence-related pattern. It enables us to implement our business code on a higher abstraction level.
- **Reduced boilerplate code:** It provides the default

implementation for each method by its repository interfaces. It means that there is no longer need to implement read and write operations.

- **Generated Queries:** Another feature of Spring Data JPA is the **generation of database queries** based on the method name. If the query is not too complex, we need to define a method on our repository interface with the name that starts with **findBy**. After defining the method, Spring parses the method name and creates a query for it. For example:

```
1 public interface EmployeeRepository extends  
  CrudRepository<Employee, Long>  
2 {  
3     Employee findByName(String name);  
4 }
```

In the above example, we extend the **CrudRepository** that uses two generics: **Employee** and **Long**. The Employee is the **entity** that is to be managed, and **Long** is the data type of primary key

Spring internally generates a **JPQL** (Java Persistence Query Language) query based on the method name. The query is derived from the method signature. It sets the bind parameter value, execute the query, and returns the result.

There are some other features are as follows:

- It can integrate custom repository code.
- It is a powerful repository and custom object-mapping abstraction.
- It supports transparent auditing.
- It implements a domain base class that provides basic properties.
- It supports several modules such as Spring Data JPA, Spring Data MongoDB, Spring Data REST, Spring Data Cassandra, etc.

Spring Data Repository

Spring Data JPA provides **three** repositories are as follows:

- **CrudRepository:** It offers standard **create, read, update,** and **delete** It contains method like **findOne(), findAll(), save(), delete(),** etc.
- **PagingAndSortingRepository:** It extends the **CrudRepository** and adds the **findAll** methods. It allows us to **sort** and **retrieve** the data in a paginated way.
- **JpaRepository:** It is a **JPA specific repository** It is defined in **Spring Data Jpa**. It extends the both **repository CrudRepository** and **PagingAndSortingRepository**. It adds the JPA-specific methods, like **flush()** to trigger a flush on the persistence context.

```
1 <dependency>
2 <groupId>org.springframework.data</groupId>
3 <artifactId>spring-data-jpa</artifactId>
4 <version>2.2.3.RELEASE</version>
5 </dependency>
```

Spring Boot Starter Data JPA

Spring Boot provides **spring-boot-starter-data-jpa** dependency to connect Spring application with relational database efficiently. The **spring-boot-starter-data-jpa** internally uses the **spring-boot-jpa** dependency (since Spring Boot version 1.5.3).

```
1 <dependency>
2 <groupId>org.springframework.boot</groupId>
```

```
3 <artifactId>spring-boot-starter-data-jpa</artifactId>
4 <version>2.2.2.RELEASE</version>
5 </dependency>
```

The databases are designed with tables/relations. Earlier approaches (JDBC) involved writing SQL queries. In the JPA, we will store the data from objects into table and vice-versa. However, JPA evolved as a result of a different thought process.

Before JPA, ORM was the term more commonly used to refer to these frameworks. It is the reason Hibernate is called the ORM framework.

JPA allows us to map application classes to table in the database.

- **Entity Manager:** Once we define the mapping, it handles all the interactions with the database.
- **JPQL (Java Persistence Query Language):** It provides a way to write queries to execute searches against entities. It is different from the SQL queries. JPQL queries already understand the mapping that is defined between entities. We can add additional conditions if required.
- **Criteria API:** It defines a Java-based API to execute searches against the database.

Hibernate vs. JPA

Hibernate is the implementation of JPA. It is the most popular ORM framework, while JPA is an API that defines the specification. Hibernate understands the mapping that we add between objects and tables. It ensures that data is retrieved/stored from the database based on the mapping. It also provides additional features on the top of the JPA.

Spring Boot JPA Example

In this example, we will use spring-boot-starter-data-jpa

dependency to create a connection with the H2 database.

Step 1: Open spring Initializr <https://start.spring.io/>.

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

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

Step 4: Add the dependencies: **Spring Web**, **Spring Data JPA**, and **H2 Database**.

Step 5: Click on the **Generate** button. When we click on the Generate button, it wraps the project in **Jar** file and downloads it to the local system.

The screenshot shows the Spring Initializr web application interface. The left sidebar contains navigation links: Project, Language, Spring Boot, Project Metadata, and Dependencies. The main content area is divided into sections: Project (Maven Project selected), Language (Java selected), Spring Boot (2.2.2 selected), Project Metadata (Group: com.javatpoint, Artifact: spring-boot-jpa-example), and Dependencies (3 selected). The Dependencies section shows a search bar and a list of selected dependencies: Spring Data JPA and Spring Web. The Generate button is highlighted with a red box.

Spring Initializr
Bootstrap your application

Project: **Maven Project** | Gradle Project

Language: **Java** | Kotlin | Groovy

Spring Boot: 2.2.3 (SNAPSHOT) | **2.2.2** | 2.1.12 (SNAPSHOT) | 2.1.11

Project Metadata

Group: **com.javatpoint**

Artifact: **spring-boot-jpa-example**

> Options

Dependencies: 3 selected

Search dependencies to add: Web, Security, JPA, Actuator, Devtools...

Selected dependencies:

- Spring Data JPA**
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate. ✓
- Spring Web**

© 2013-2019 Pivotal Software
start.spring.io is powered by
[Spring Initializr](#) and [Pivotal Web Services](#)

Generate - Ctrl + G | Explore - Ctrl + Space | Share...

Step 6: Extract the Jar file and paste it into the STS workspace.

Step 7: Import the project folder into STS.

File -> Import -> Existing Maven Projects -> Browse -> Select the folder spring-boot-jpa-example -> Finish

It takes some time to import.

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

Step 9: Create a Controller class with the name **ControllerDemo** in the package **com.javatpoint.controller**.

ControllerDemo.java

```
1 package com.javatpoint.controller;
2 import org.springframework.stereotype.Controller;
3
4     i           m           p           o           r           t
   org.springframework.web.bind.annotation.RequestMapping;
5 @Controller
6 public class ControllerDemo
7 {
8     @RequestMapping("/")
9     public String home()
10 {
11     return "home.jsp";
12 }
```

Step 10: Create another package with the name

com.javatpoint.model in the folder **src/main/java**.

Step 11: Create a class with the name **User** in the package **com.javatpoint.model**.

User.java

```
1 package com.javatpoint.model;
2 import javax.persistence.Entity;
3 import javax.persistence.Id;
4 import javax.persistence.Table;
5 @Entity
6 @Table(name="userdata")
7 public class User
8 {
9     @Id
10    private int id;
11    private String username;
12    public int getId()
13    {
14        return id;
15    }
16    public void setId(int id)
17    {
18        this.id = id;
19    }
20    public String getUname()
21    {
22        return username;
23    }
24    public void setUname(String username)
25    {
26        this.username = username;
27    }
28    @Override
29    public String toString()
30    {
```

```
31 return "User [id=" + id + ", uname=" + username + "];  
32 }  
33 }
```

Now we need to Configure the H2 database.

Step 12: Open the **application.properties** file and configure the following things: **port**, **enable the H2 console**, **datasource**, and **URL**.

application.properties

```
1  server.port=8085  
2  spring.h2.console.enabled=true  
3  spring.datasource.platform=h2  
4  spring.datasource.url=jdbc:h2:mem:javatpoint
```

Step 13: Create a **SQL** file in the folder **src/main/resources**.

Right-click on the folder **src/main/resources** -> New -> File -> Provide the **File name** -> Finish

We have provided the file name **data.sql** and insert the following data into it.

data.sql

```
1  insert into userdata values(101,'Tom');  
2  insert into userdata values(102,'Andrew');  
3  insert into userdata values(103,'Tony');  
4  insert into userdata values(104,'Bob');  
5  insert into userdata values(105,'Sam');
```

Step 14: Create a folder with the name **webapp** in the **src** folder.

Step 15: Create a JSP file with the name that we have returned in the **ControllerDemo**. In the ControllerDemo.java, we have returned **home.jsp**.

home.jsp

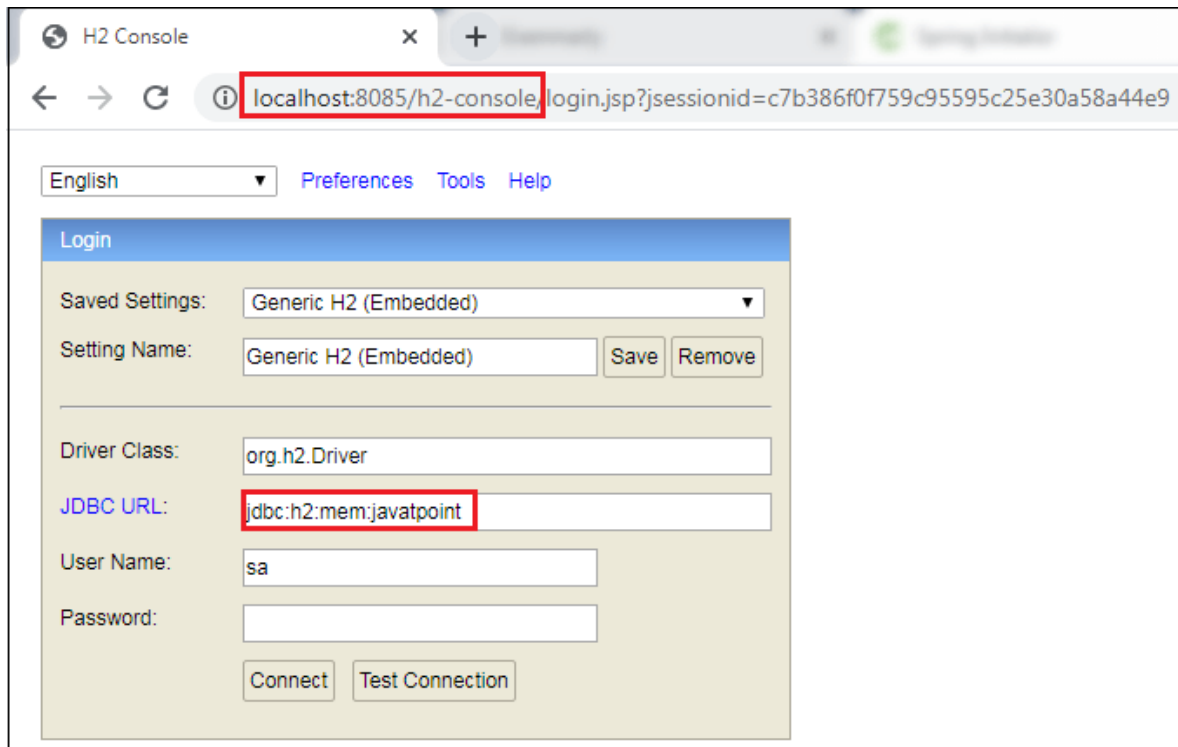
```
1 <%@ page language="java" contentType="text/
  html; charset=ISO-8859-1"
2 pageEncoding="ISO-8859-1"%>
3 <!DOCTYPE html>
4 <html>
5 <head>
6 <meta charset="ISO-8859-1">
7 <title>Insert title here</title>
8 </head>
9 <body>
10 <form action="addUser">
11 ID :<br />
12 <input type="text" name="t1"><br />
13 User name :<br />
14 <input type="text" name="t2"><br />
15 <input type="submit" value="Add">
16 </form>
17 </body>
18 </html>
```

Step 16: Run the **SpringBootJpaExampleApplication.java** file. We can see in the console that our application is successfully running on port **8085**.

```
Tomcat started on port(s): 8085 (http) with context path ''
Started SpringBootJpaExampleApplication in 42.692 seconds (JVM running for 49.673)
Initializing Spring DispatcherServlet 'dispatcherServlet'
Initializing Servlet 'dispatcherServlet'
Completed initialization in 80 ms
```

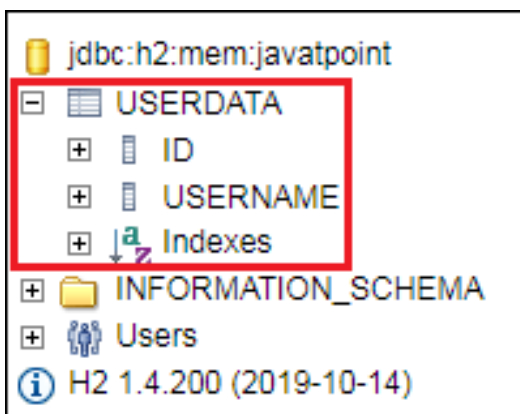
Step 17: Open the browser and invoke the URL <http://localhost:8085/h2-console/>. It shows the Driver Class, JDBC URL that we have configured in the **application.properties**

file, and the default User Name sa.



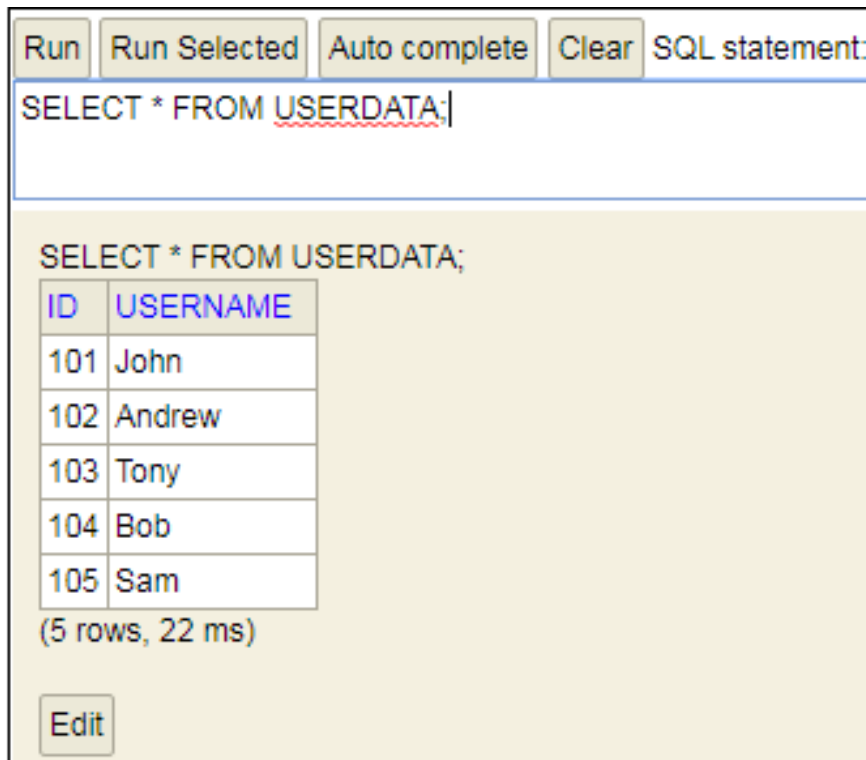
We can also test the connection by clicking on the **Test Connection** button. If the connection is successful, it shows a message Test Successful.

Step 18: Click on the **Connect** button. It shows the structure of the table userdata that we have defined in the **User.java**.



Step 19: Execute the following query to see the data that we have inserted in the **data.sql** file.


```
1 SELECT * FROM USERDATA;
```



The screenshot shows a web-based SQL interface. At the top, there are four buttons: "Run", "Run Selected", "Auto complete", and "Clear", followed by the label "SQL statement:". Below this is a text input field containing the SQL query "SELECT * FROM USERDATA;" with a red squiggly underline under "USERDATA". Below the input field, the same query is displayed above a table of results. The table has two columns, "ID" and "USERNAME", and five rows of data. Below the table, it says "(5 rows, 22 ms)". At the bottom left, there is an "Edit" button.

ID	USERNAME
101	John
102	Andrew
103	Tony
104	Bob
105	Sam

(5 rows, 22 ms)

Edit

Spring Boot Packaging

In the J2EE application, modules are packed as **JAR**, **WAR**, and **EAR**. It is the compressed file formats that is used in the J2EE. J2EE defines three types of archives:

- WAR
- JAR
- EAR



WAR

WAR stands for **Web Archive**. WAR file represents the web application. Web module contains servlet classes, JSP files, HTML files, JavaScripts, etc. are packaged as a JAR file with **.war** extension. It contains a special directory called **WEB-INF**.

WAR is a module that loads into a web container of the Java Application Server. The Java Application Server has **two** containers: **Web Container** and **EJB Container**.

The **Web Container** hosts the web applications based on Servlet API and JSP. The web container requires the web

module to be packaged as a WAR file.

An **EJB Container** hosts Enterprise Java beans based on EJB API. It requires EJB modules to be packaged as a JAR file. It contains an **ejb-jar.xml** file in the **META-INF** folder.

The advantage of the WAR file is that it can be deployed easily on the client machine in a Web server environment. To execute a WAR file, a Web server or Web container is required. For example, Tomcat, Weblogic, and Websphere.

JAR

JAR stands for **Java Archive**. An EJB (Enterprise Java Beans) module that contains bean files (class files), a manifest, and EJB deployment descriptor (XML file) are packaged as JAR files with the extension **.jar**. It is used by software developers to distribute Java classes and various metadata.

In other words, A file that encapsulates one or more Java classes, a manifest, and descriptor is called JAR file. It is the lowest level of the archive. It is used in J2EE for packaging EJB and client-side Java Applications. It makes the deployment easy.

EAR

EAR stands for **Enterprise Archive**. EAR file represents the enterprise application. The above two files are packaged as a JAR file with the **.ear** extension. It is deployed into the Application Server. It can contain multiple EJB modules (JAR) and Web modules (WAR). It is a special JAR that contains an **application.xml** file in the **META-INF** folder.

