# 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.

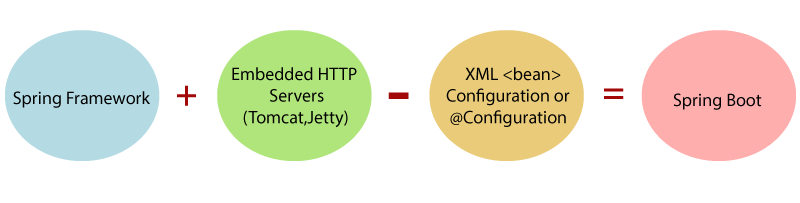
Spring Boot is a Spring module that provides the RAD (Rapid Application Development) feature to the Spring framework.

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.

* 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.

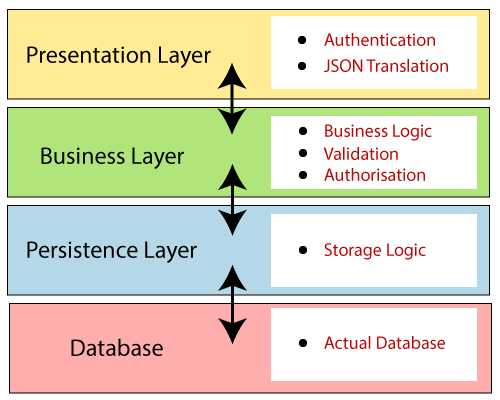
**Security**

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 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**,

There are **four** layers in Spring Boot.



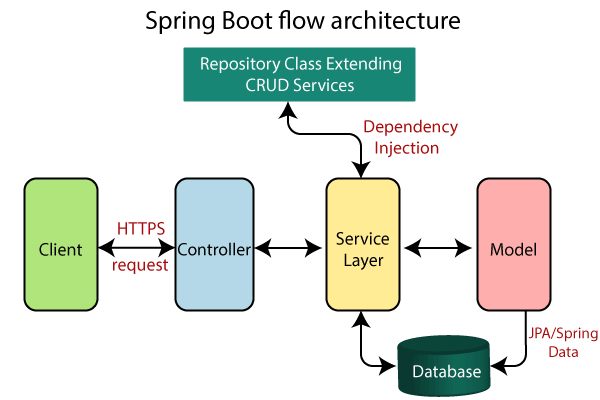
**Presentation Layer:** The presentation layer handles the HTTP requests, translates the JSON parameter to object, and authenticates the request and transfer it to the business layer. In short, it consists of **views** i.e., frontend part.

**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.

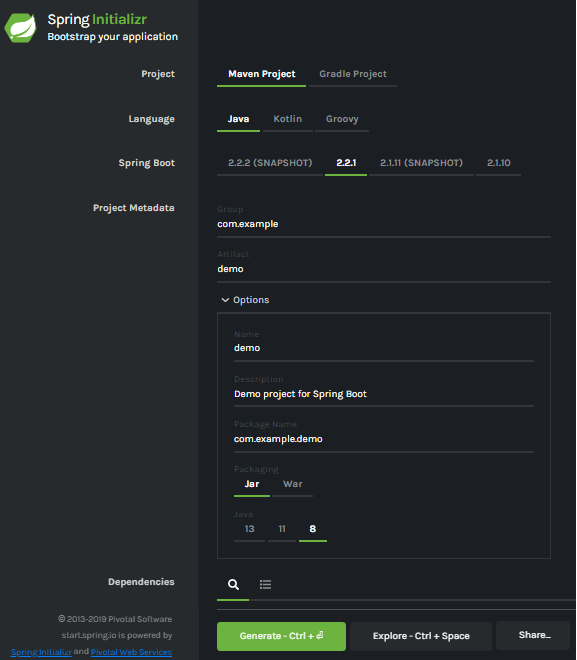
**Database Layer:** In the database layer, **CRUD** (create, retrieve, update, delete) operations are performed.

## ring 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.



## 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.

### Installing STS

**Step 1:** Download Spring Tool Suite from <https://spring.io/tools3/sts/all>.

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

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

# 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.java** 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:

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

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

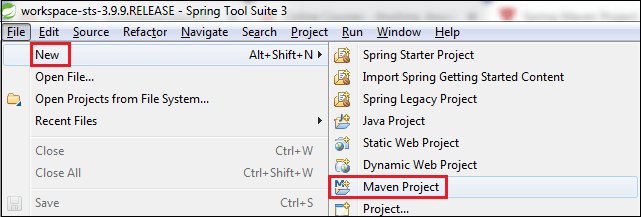
The application runs successfully.

# 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.java** 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.java;
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.java**</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>

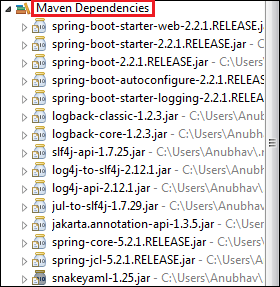
**Step 7:** In order to make a Spring Boot Project, we need to configure it. So, we are adding **spring boot starter parent** dependency in **pom.xml** file. Parent is used to declare that our project is a child to this parent project.

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.java**</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.java**.

ADVERTISEMENT

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.

# 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.java")
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.java;
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.java;
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.

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.

#### **Note: We have used all the above annotations in our**[**RESTful Web**](https://www.javatpoint.com/restful-web-services-tutorial) **Services with real-world examples**

# Spring Boot Dependency Management

Spring Boot manages dependencies and configuration automatically. Each release of Spring Boot provides a list of dependencies that it supports. The list of dependencies is available as a part of the **Bills of Materials** (spring-boot-dependencies) that can be used with **Maven**. So, we need not to specify the version of the dependencies in our configuration. Spring Boot manages itself. Spring Boot upgrades all dependencies automatically in a consistent way when we update the Spring Boot version.

## Advantages of Dependency Management

* It provides the centralization of dependency information by specifying the Spring Boot version in one place. It helps when we switch from one version to another.
* It avoids mismatch of different versions of Spring Boot libraries.
* We only need to write a library name with specifying the version. It is helpful in multi-module projects.

#### **Note: Spring Boot also allows overriding of dependencies version, if required.**

## Maven Dependency Management System

The Maven project inherits the following features from **spring-boot-starter-parent:**

* The default **Java compiler version**
* **UTF-8** source encoding
* It inherits a **Dependency Section** from the spring-boot-dependency-pom. It manages the version of common dependencies. It ignores the **<version>** tag for that dependencies.
* Dependencies, inherited from the spring-boot-dependencies POM
* Sensible **resource filtering**
* Sensible **plugin configuration**

### Inheriting Starter Parent

The following **spring-boot-starter-parent** inherits automatically when we configure the project.

1. <parent>
2. <groupId>org.springframework.boot</groupId>
3. <artifactId>spring-boot-starter-parent</artifactId>
4. <version>2.2.2.BUILD-SNAPSHOT</version>      <!-- lookup parent from repository -->
5. <relativePath/>
6. </parent>

#### **Note: In the above dependency, we have specified only the Spring Boot version. If we want to add additional starters, simply remove the <version> tag. Similarly, we can also override the individual dependency by overriding a property in our project.**

For example, if we want to add another dependency with the same artifact that we have injected already, inject that dependency again inside the **<properties>** tag to override the previous one

### Changing the Java version

We can also change the Java version by using the **<java.version>** tag.

1. <properties>
2. <java.version>1.8</java.version>
3. </properties>

### Adding Spring Boot Maven Plugin

We can also **add Maven plugin** in our **pom.xml** file. It wraps the project into an executable **jar** file.

1. <build>
2. <plugins>
3. <plugin>
4. <groupId>org.springframework.boot</groupId>
5. <artifactId>spring-boot-maven-plugin</artifactId>
6. </plugin>
7. </plugins>
8. </build>

### Spring Boot without Parent POM

If we don't want to use **spring-boot starter-parent** dependency, but still want to take the advantage of the dependency management, we can use **<scope>** tag, as follows:

#### **Note: It does not maintain the plugin management.**

1. <dependencyManagement>
2. <dependencies>
3. <dependency><!-- Import dependency management from Spring Boot -->
4. <groupId>org.springframework.boot</groupId>
5. <artifactId>spring-boot-dependencies</artifactId>
6. <version>2.2.2.RELEASE</version>
7. <type>pom</type>
8. <scope>**import**</scope>
9. </dependency>
10. </dependencies>
11. </dependencyManagement>

The above dependency does not allow overriding. To achieve the overriding, we need to add an entry inside the **<dependencyManagement>**tag of our project before the spring-boot-dependencies entry.

For example, to upgrade another **spring-data-releasetrain,**add the following dependency in the pom.xml file.

1. <dependencyManagement>
2. <dependencies>
3. <!--Override Spring Data release train-->
4. <dependency>
5. <groupId>org.springframework.data</groupId>
6. <artifactId>spring-data-releasetrain</artifactId>
7. <version>Fowler-SR2</version>
8. <type>pom</type>
9. <scope>**import**</scope>
10. </dependency>
11. <dependency>
12. <groupId>org.springframework.boot</groupId>
13. <artifactId>spring-boot-dependencies</artifactId>
14. <version>2.2.2.RELEASE</version>
15. <type>pom</type>
16. <scope>**import**</scope>
17. </dependency>
18. </dependencies>
19. </dependencyManagement>

# 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

## Application Properties Table

The following tables provide a list of common Spring Boot properties

| | **Property** | **Default Values** | **Description** | | --- | --- | --- | | Debug | false | It enables debug logs. | | spring.application.name |  | It is used to set the application name. | | spring.application.admin.enabled | false | It is used to enable admin features of the application. | | spring.config.name | application | It is used to set config file name. | | spring.config.location |  | It is used to config the file name. | | server.port | 8080 | Configures the HTTP server port | | server.servlet.context-path |  | It configures the context path of the application. | | logging.file.path |  | It configures the location of the log file. | | spring.banner.charset | UTF-8 | Banner file encoding. | | spring.banner.location | classpath:banner.txt | It is used to set banner file location. | | logging.file |  | It is used to set log file name. For example, data.log. | | spring.application.index |  | It is used to set application index. | | spring.application.name |  | It is used to set the application name. | | spring.application.admin.enabled | false | It is used to enable admin features for the application. | | spring.config.location |  | It is used to config the file locations. | | spring.config.name | application | It is used to set config the file name. | | spring.mail.default-encoding | UTF-8 | It is used to set default MimeMessage encoding. | | spring.mail.host |  | It is used to set SMTP server host. For example, smtp.example.com. | | spring.mail.password |  | It is used to set login password of the SMTP server. | | spring.mail.port |  | It is used to set SMTP server port. | | spring.mail.test-connection | false | It is used to test that the mail server is available on startup. | | spring.mail.username |  | It is used to set login user of the SMTP server. | | spring.main.sources |  | It is used to set sources for the application. | | server.address |  | It is used to set network address to which the server should bind to. | | server.connection-timeout |  | It is used to set time in milliseconds that connectors will wait for another HTTP request before closing the connection. | | server.context-path |  | It is used to set context path of the application. | | server.port | 8080 | It is used to set HTTP port. | | server.server-header |  | It is used for the Server response header (no header is sent if empty) | | server.servlet-path | / | It is used to set path of the main dispatcher servlet | | server.ssl.enabled |  | It is used to enable SSL support. | | spring.http.multipart.enabled | True | It is used to enable support of multi-part uploads. | | spring.servlet.multipart.max-file-size | 1MB | It is used to set max file size. | | spring.mvc.async.request-timeout |  | It is used to set time in milliseconds. | | spring.mvc.date-format |  | It is used to set date format. For example, dd/MM/yyyy. | | spring.mvc.locale |  | It is used to set locale for the application. | | spring.social.facebook.app-id |  | It is used to set application's Facebook App ID. | | spring.social.linkedin.app-id |  | It is used to set application's LinkedIn App ID. | | spring.social.twitter.app-id |  | It is used to set application's Twitter App ID. | | security.basic.authorize-mode | role | It is used to set security authorize mode to apply. | | security.basic.enabled | true | It is used to enable basic authentication. | | Spring.test.database.replace | any | Type of existing DataSource to replace. | | Spring.test.mockmvc.print | default | MVC Print option | | spring.freemaker.content-type | text/html | Content Type value | | server.server-header |  | Value to use for the server response header. | | spring.security.filter.dispatcher-type | async, error, request | Security filter chain dispatcher types. | | spring.security.filter.order | -100 | Security filter chain order. | | spring.security.oauth2.client.registration.\* |  | OAuth client registrations. | | spring.security.oauth2.client.provider.\* |  | OAuth provider details. | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

**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**.

| **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

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.

# Spring Boot Starter Web

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

* It is compatible for web development
* Auto configuration

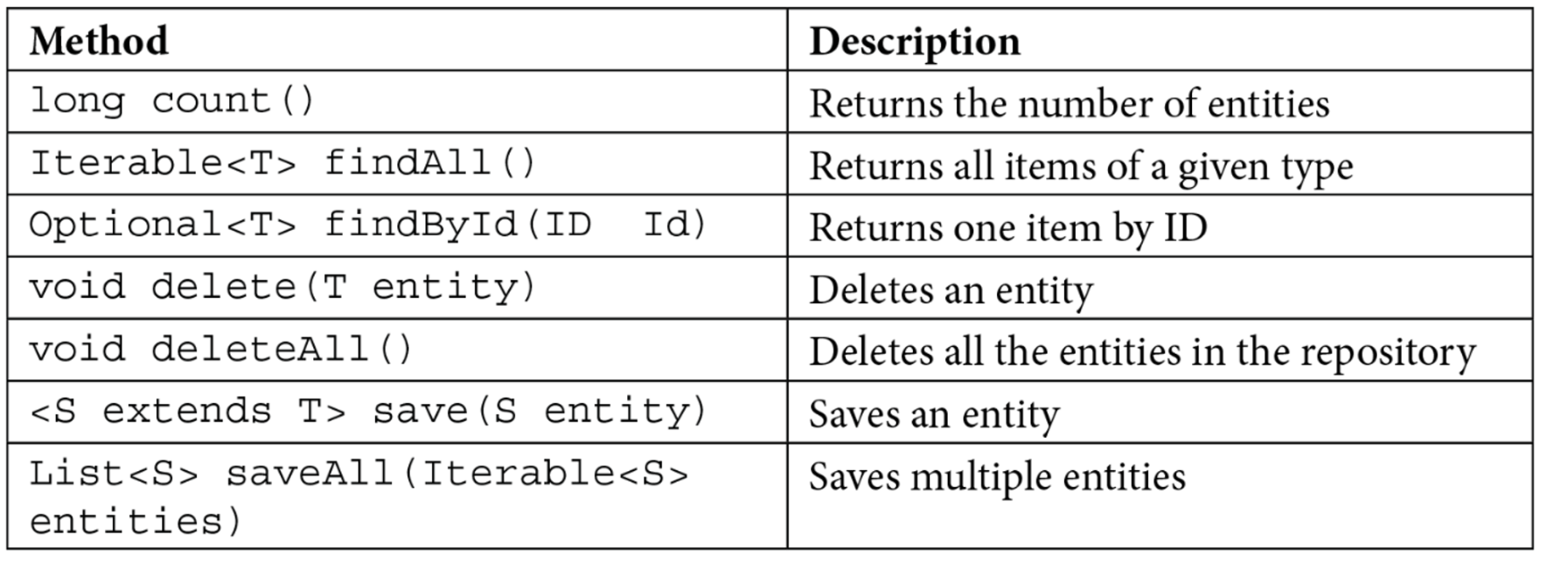
## Spring Data JPA

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 Repository

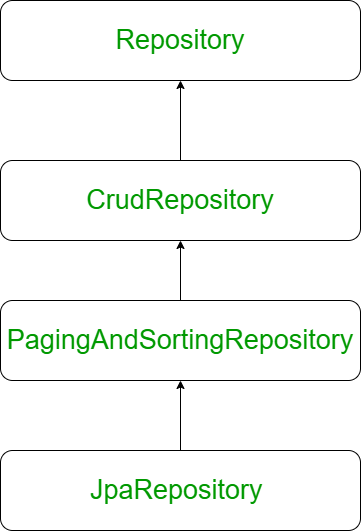
Spring Data JPA provides **three** repositories are as follows:

* **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.

CrudRepository and PagingAndSortingRepository belong to Spring Data Commons whereas JpaRepository belongs to Spring Data JPA.



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.java**.

**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.

**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.java.controller**in the folder **src/main/java**.

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

**ControllerDemo.java**

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

**Step 10:**Create another package with the name **com.java.model**in the folder **src/main/java.**

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

**User.java**

1. package com.java.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.plateform=h2
4. spring.datasource.url=jdbc:h2:mem:java

**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');

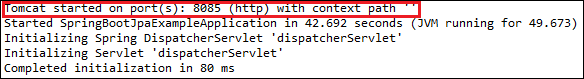
**Step 14:** Create a folder with the name **webapp** in the **src** folder.

**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**.



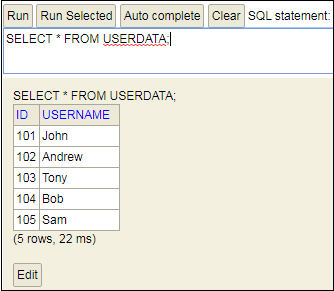
**Step 17:** Open the browser and invoke the URL http://localhost:8085/h2-console/. It shows the Driver Class, JDBC URL that we have configured in the **application.properties** file, and the default User Name sa.

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;



# Spring Boot Starter Actuator

## Spring Boot Actuator

**Spring Boot Actuator** is a sub-project of the Spring Boot Framework. It includes a number of additional features that help us to monitor and manage the Spring Boot application. It contains the actuator endpoints (the place where the resources live). We can use **HTTP** and **JMX** endpoints to manage and monitor the Spring Boot application. If we want to get production-ready features in an application, we should use the S**pring Boot actuator.**

### Spring Boot Actuator Features

There are **three** main features of Spring Boot Actuator:

* **Endpoints**
* **Metrics**
* **Audit**

**Endpoint:** The actuator endpoints allows us to monitor and interact with the application. Spring Boot provides a number of built-in endpoints. We can also create our own endpoint. We can enable and disable each endpoint individually. Most of the application choose **HTTP**, where the Id of the endpoint, along with the prefix of **/actuator,**is mapped to a URL.

For example, the **/health** endpoint provides the basic health information of an application. The actuator, by default, mapped it to **/actuator/health**.

**Metrics:** Spring Boot Actuator provides dimensional metrics by integrating with the**micrometer**. The micrometer is integrated into Spring Boot. It is the instrumentation library powering the delivery of application metrics from Spring. It provides vendor-neutral interfaces for **timers, gauges, counters, distribution summaries,** and **long task timers** with a dimensional data model.

**Audit:** Spring Boot provides a flexible audit framework that publishes events to an **AuditEventRepository.** It automatically publishes the authentication events if spring-security is in execution.

## Enabling Spring Boot Actuator

We can enable actuator by injecting the dependency **spring-boot-starter-actuator** in the pom.xml file.

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

## Spring Boot Actuator Endpoints

The actuator endpoints allow us to monitor and interact with our Spring Boot application. Spring Boot includes number of built-in endpoints and we can also add custom endpoints in Spring Boot application.

The following table describes the widely used endpoints.

| **Id** | **Usage** | **Default** |
| --- | --- | --- |
| actuator | It provides a hypermedia-based **discovery page** for the other endpoints. It requires Spring HATEOAS to be on the classpath. | True |
| auditevents | It exposes audit events information for the current application. | True |
| autoconfig | It is used to display an auto-configuration report showing all auto-configuration candidates and the reason why they 'were' or 'were not' applied. | True |
| beans | It is used to display a complete list of all the Spring beans in your application. | True |
| configprops | It is used to display a collated list of all @ConfigurationProperties. | True |
| dump | It is used to perform a thread dump. | True |
| env | It is used to expose properties from Spring's ConfigurableEnvironment. | True |
| flyway | It is used to show any Flyway database migrations that have been applied. | True |
| health | It is used to show application health information. | False |
| info | It is used to display arbitrary application info. | False |
| loggers | It is used to show and modify the configuration of loggers in the application. | True |
| liquibase | It is used to show any Liquibase database migrations that have been applied. | True |
| metrics | It is used to show metrics information for the current application. | True |
| mappings | It is used to display a collated list of all @RequestMapping paths. | True |
| shutdown | It is used to allow the application to be gracefully shutdown. | True |
| trace | It is used to display trace information. | True |

For Spring MVC, the following additional endpoints are used.

## Spring Boot actuator properties

Spring Boot enables security for all actuator endpoints. It uses **form-based** authentication that provides **user Id** as the user and a randomly generated **password**. We can also access actuator-restricted endpoints by customizing basicauth security to the endpoints. We need to override this configuration by **management.security.roles** property. For example:

1. management.security.enabled=true
2. management.security.roles=ADMIN
3. security.basic.enabled=true
4. security.user.name=admin
5. security.user.passowrd=admin

### Spring Boot Actuator Example

Let's understand the concept of the actuator through an example.

**Step 1:** Open Spring Initializr <https://start.spring.io/> and create a **Maven** project.

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

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

**Step 4:** Add the following dependencies: **Spring Web, Spring Boot Starter Actuator,** and **Spring Data Rest HAL Browser**.

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

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

**Step 7:** Import the project folder.

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

**Step 8:** Create a Controller class. We have created the controller class with the name DemoRestController.

**DemoRestController.java**

1. package com.java;
2. import org.springframework.web.bind.annotation.GetMapping;
3. import org.springframework.web.bind.annotation.RestController;
4. @RestController
5. public class DemoRestController
6. {
7. @GetMapping("/hello")
8. public String hello()
9. {
10. return "Hello User!";
11. }
12. }

**Step 9:** Open the **application.properties** file and disable the security feature of the actuator by adding the following statement.

**application.properties**

1. management.security.enabled=false

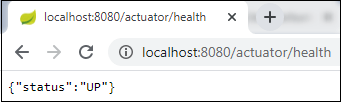
**Step 10:** Run the **SpringBootActuatorExampleApplication.java** file.

**Step 11:** Open the browser and invoke the URL http://localhost:8080/actuator. It returns the following page:

1. {"\_links":{"self":{"href":"http://localhost:8080/actuator","templated":false},"health":{"href":"http://localhost:8080/actuator/health","templated":false},"health-path":{"href":"http://localhost:8080/actuator/health/{\*path}","templated":true},"info":{"href":"http://localhost:8080/actuator/info","templated":false}}}

The application runs on port 8080 by default. Once the actuator has started, we can see the list of all the endpoints exposed over HTTP.

Let's invoke the **health** endpoint by invoking the URL http://localhost:8080/actuator/health. It denotes the status **UP**. It means the application is healthy and running without any interruption.



Similarly, we can invoke other endpoints that helps us to monitor and manage the Spring Boot application.

# Spring Boot Starter Test

The **spring-boot-starter-test** is the primary dependency for the test. It contains the majority of elements required for our tests.

There are several different types of tests that we can write to help test and automate the health of an application. Before starting any testing, we need to integrate the testing framework.

With Spring Boot, we need to add **starter** to our project, for testing we only need to add the **spring-boot-starter-test** dependency.

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

It pulls all the dependencies related to test. After adding it, we can build up a simple unit test. We can either create the Spring Boot project through IDE or generate it using Spring Initializr.

#### **Note: If you are adding test dependency manually, add it to the bottom of the pom.xml file.**

In the above dependency, one thing to be noticed that it includes the scope of test **<scope>test</scope>.** It means when the application is bundled and packaged for deployment, any dependency that is declared with the test scopes is ignored. The test scope dependencies are only available when running in the development and Maven test modes.

When we create a simple Spring Boot application, by default, it contains the test dependency in the pom.xml file and **ApplicationNameTest.java** file under in the folder **src/test/java.**

Let's create a simple maven project.

## Spring Boot Starter Test Example

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

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

**Step 3:** Add the Spring Web dependency.

**Step 4:** Click on the **Generate** button. When we click on the Generate button, it wraps all the specifications related to the project and downloads a **Jar** file to our local system.

**Step 5:** Extract the downloaded Jar file.

**Step 6:** Import the folder to STS. It takes some time to import.

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

After importing the project, we can see the following project directory in the Package Explorer section of the STS.

We can see in the above directory that it contains a test file named **SpringBootTestExampleApplicationTest.java** in the folder **src/test/java.**

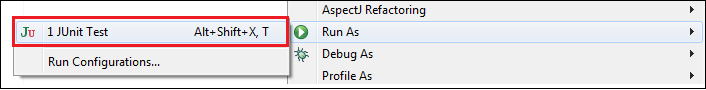
**SpringBootTestExampleApplicationTest.java**

1. **package** com.java.springboottestexample;
2. **import** org.junit.jupiter.api.Test;
3. **import** org.springframework.boot.test.context.SpringBootTest;
4. @SpringBootTest
5. **class** SpringBootTestExampleApplicationTests
6. {
7. @Test
8. **void** contextLoads()
9. {
10. }
11. }

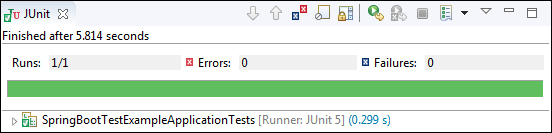
The above code implements **two** annotation by default: **@SpringBootTest,** and **@Test.**

* **@SpringBootTest:** It applies on a Test Class that runs Spring Boot based tests. It provides the following features over and above the regular Spring TestContext Framework:
  + It uses **SpringBootContextLoader** as the default ContextLoader if no specific @ContextConfiguration(loader=...) is defined.
  + It automatically searches for a **@SpringBootConfiguration** when nested @Configuartion is not used, and no explicit classes are specified.
  + It provides support for different **WebEnvironment** modes.
  + It registers a **TestRestTemplate** or WebTestClient bean for use in web tests that are using the webserver.
  + It allows application arguments to be defined using the **args attribute.**

**Step 7:** Open the **SpringBootTestExampleApplicationTest.java** file and run it as **Junit Test.**



When we run the above code, it displays the following



# 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

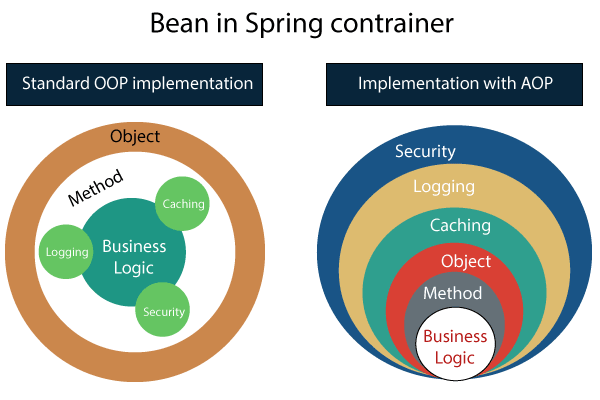
* **Aspect:** An aspect is a module that encapsulates **advice** and **pointcuts** and provides **cross-cutting** An application can have any number of aspects. We can implement an aspect using regular class annotated with **@Aspect** annotation.
* **Pointcut:** A pointcut is an expression that selects 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>**