**Spring Boot**

Spring is widely used for creating scalable applications.

(*For web applications Spring provides* ***Spring MVC*** *which is a widely used module of spring which is used to create scalable web applications. But main disadvantage of spring projects is that* ***configuration*** *is really time-consuming and can be a bit overwhelming for the new developers. Making the application production-ready takes some time if you are new to the spring.*)

Spring Boot is an open-source Java-based framework used to create a Microservice and is a module that provides the RAD (Rapid Application Development) feature to the Spring framework, developed by Pivotal Team. Spring Boot is built on the top of the Spring and contains all the features of Spring framework. It is easy to create a stand-alone and production ready spring applications using Spring Boot. Spring Boot contains a comprehensive infrastructure support for developing a microservice and enables you to develop enterprise-ready applications that you can “just run”.

*(And is becoming favourite of developer’s these days because of it’s a rapid production-ready environment which enables the developers to directly focus on the logic instead of struggling with the configuration and set up.)*

The basic difference in bootstrapping of an application in Spring and Spring Boot lies with the **servlet**. Spring uses either the **web.xml** or **SpringServletContainerInitializer** as its bootstrap entry point. On the other hand, Spring Boot uses only Servlet 3 features to bootstrap an application.

**Features of Spring Boot**

Spring Boot is built on the top of the conventional spring framework. So, it provides all the features of Spring and is yet easier to use than Spring.

* It allows to avoid heavy configuration of XML which is present in Spring:

Unlike the **Spring MVC Project**, in Spring Boot everything is auto-configured. We just need to use proper configuration for utilizing a particular functionality.

Example: *If we want to use hibernate(ORM) then we can just add* ***@Table*** *annotation above model/entity class and add* ***@Column*** *annotation to map it to table and columns in the database.*

* It provides easy maintenance and creation of REST end points:

Creating a REST API is very easy in Spring Boot. Just the annotation **@RestController** and **@RequestMapping(/endPoint)** over the controller class does the work.

* It includes embedded Tomcat-server:

Unlike **Spring MVC project** where we have to manually add and install the tomcat server, Spring Boot comes with an embedded Tomcat server, so that the applications can be hosted on it.

* Deployment is very easy, **war** and **jar** file can be easily deployed in the tomcat server:

**war** or **jar** files can be directly deployed on the Tomcat Server and Spring Boot provides the facility to convert our project into **war** or **jar** files. Also, the instance of Tomcat can be run on the *cloud* as well.

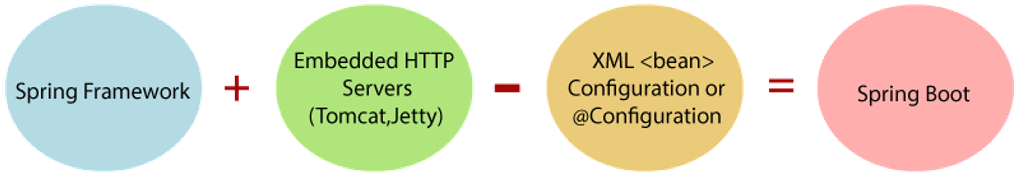
* Microservice Based Architecture:

Microservice, as the name suggests is the name given to a module/service which focuses on a single type of feature, exposing an API (*Application Peripheral Interface*).

Example : In a Hospital Management System.

* In case of monolithic systems, there will be a single code containing all the features which are very tough to maintain on a huge scale.
* But in the microservice-based system, each feature can be divided into smaller subsystems like service to handle **patient registration**, service to handle **database management**, service to handle **billing** etc.

Microservice based system can be easily migrated as only some services need to be altered which also makes debugging and deployment easy. Also, each service can be integrated and can be made in different technologies suited to them.



**Evolution of Spring Boot**

Spring Boot came into existence when in October 2012, a client, Mike Youngstrom made a JIRA request asking for bootstrapping the **Spring Framework** so that it can be quickly started. And hence in early **2013**, Spring Boot was made.

* In April 2014, Spring Boot 1.0v was created followed by various versions.
* Spring Boot 1.1 on June 2014,
* 1.2 in March 2015,
* 1.3 in December 2016,
* 1.4 in January 2017 and
* Spring Boot 1.5 on February 2017.

**Spring Boot Architecture**

Spring Boot Architecture : Different layers and classes present in it.

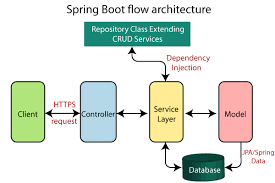
There are **four** main layers in Spring Boot

1. Presentation Layer : As the name suggests, it consists of views (i.e. frontend part)
2. Data Access Layer : CRUD (create, retrieve, update, delete) operations on the database comes

under this category.

1. Service Layer : This consist of service classes and uses services provided by data access layers.
2. Integration Layer : It consists of different web services *(any service available over the internet and uses*

*XML messaging system)*.



* Important classes : utility classes, validator classes and view classes.
* Eases dependency management
* All the services provided by the classes are implemented in their corresponding classes and are retrieved by implementing the dependency on those interfaces.
* In Spring Boot, everything is auto configured; no manual configurations are needed.
* It provides a flexible way to configure Java Beans, XML configurations, and Database Transactions.
* It provides a powerful batch processing and manages REST endpoints.
* Since Spring Boot uses all the features/modules of spring-like **Spring data**, **Spring MVC** etc. so the architecture is almost the same as **spring MVC**, except for the fact that there is no need of **DAO** and **DAOImpl** classes in **Spring boot**.
* Creating a data access layer needs just a repository class instead which is implementing **CRUD** operation containing class.
* It includes Embedded **Servlet Container**. A client makes the https request (PUT/GET)
* Then it goes to controller and the controller mapped with that route as that of request handles it, and calls the service logic if required.
* Business logic is performed in the **service layer** which might be performing the logic on the data from the **database** which is mapped through **JPA** with model/entity class
* Finally, a JSP page is returned in the response if no error occurred.

**What is Microservice?** Micro Service is an architecture that allows the developers to develop and deploy services independently.

Each service running has its own process and this achieves the lightweight model to support business applications.

**Advantages** Easy deployment

Simple scalability

Compatible with Containers

Minimum configuration

Lesser production time

**Spring Boot goals** To avoid complex XML configuration in Spring

To develop a production ready Spring applications in an easier way

To reduce the development time and run the application independently

Offer an easier way of getting started with the application

**Setup Spring Boot**

1. Setup Java JDK from Oracle’s official site.
2. Download and Setup **STS** (Spring Tools Suite).
3. Start a new **spring starter project**

* Click on **File** -> **New** -> **Spring starter project**
* Fill the appropriate details and add dependency and finish.
* Edit the application properties.
* Run the main file as a Java application.

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

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

**Note** that all Spring Boot starters follow the same naming pattern spring-boot-starter- \*, where \* indicates that it is a type of the application. The following Spring Boot starters explained below for a better understanding −

Spring Boot Starter Actuator dependency is used to **monitor** and **manage your application**.

<dependency>

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

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

</dependency>

Spring Boot Starter Security dependency is used for **Spring Security**.

<dependency>

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

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

</dependency>

Spring Boot Starter web dependency is used to write a **Rest Endpoints**.

<dependency>

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

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

</dependency>

Spring Boot Starter Thymeleaf dependency is used to create a **web application**.

<dependency>

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

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

</dependency>

Spring Boot Starter Test dependency is used for writing **Test cases**.

<dependency>

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

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

</dependency>

**Auto Configuration** Spring Boot Auto Configuration automatically configures your Spring application based on the JAR dependencies you added in the project by using **@EnableAutoConfiguration** annotation.

*(Example, if MySQL database is on your classpath, but you have not configured any database connection, then Spring Boot auto-configures an in-memory database).*

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

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.EnableAutoConfiguration;

@EnableAutoConfiguration

public class **DemoApplication** {

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

SpringApplication.run(DemoApplication.class, args);

}

}

**Spring Boot Application**

The entry point of the Spring Boot application is the class contains @SpringBootApplication annotation. This class should have the **main** method to run the Spring Boot application. Spring Boot automatically scans all the components included in the project by using @ComponentScan annotation.

@SpringBootApplication annotation includes Auto- Configuration, Component Scan, and Spring Boot Configuration.

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

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class **DemoApplication** {

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

SpringApplication.run(DemoApplication.class, args);

}

}

**Component Scan**

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

import org.springframework.boot.SpringApplication;

import org.springframework.context.annotation.ComponentScan;

@ComponentScan

public class DemoApplication {

public static void main(String[] args) {

SpringApplication.run(DemoApplication.class, args);

}

}

**Creating a Spring Boot Project**

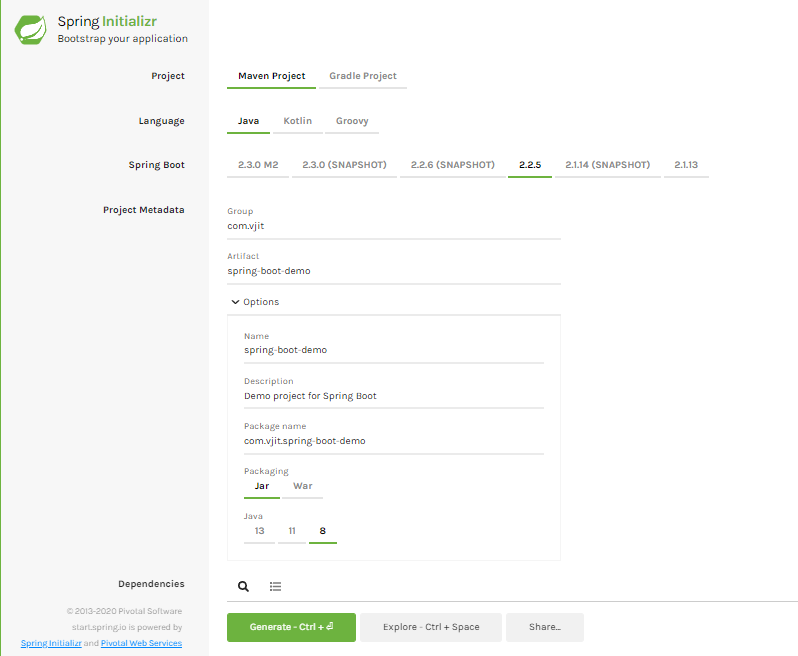
Following are the steps to create a simple Spring Boot Project.

Step 1: Open the Spring initializr https://start.spring.io.

Step 2: Provide the **Group** and **Artifact** name.

We have provided Group name **com.online** and Artifact **spring-boot-Demo**.

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

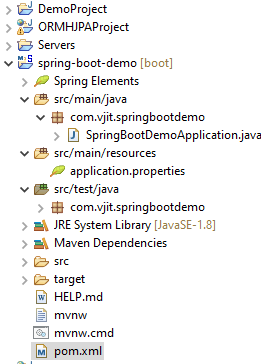
It takes some time to import the project. When the project imports successfully, we can see the project directory in the Package

Three folders were automatically created:

**src/main/java** - used to save all java source files

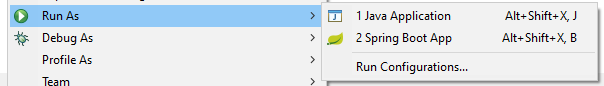
**src/main/resources** - used for templates and any other files

**src/main/test** - used for tests



Step 6: Run the SpringBootExampleApplication.java file.

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



The following image shows the application runs successfully.



**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.

**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**.

**@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.

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

**@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.

**@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.

**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.

**@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.

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

**@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.

**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.

**@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

**@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.

**@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.

**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.

**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.

<parent>

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

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

<version>2.2.2.BUILD-SNAPSHOT</version> *<!-- lookup parent from repository -->*

<relativePath/>

</parent>

*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.

<properties>

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

</properties>

**Adding Spring Boot Maven Plugin** Can also add Maven plugin in our **pom.xml** file. It wraps the project into

an executable jar file.

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</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:

<dependencyManagement>

<dependencies>

<dependency> *<!-- Import dependency management from Spring Boot -->*

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

<artifactId>spring-boot-dependencies</artifactId>

<version>2.2.2.RELEASE</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</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.

<dependencyManagement>

<dependencies>  *<!--Override Spring Data release train-->*

<dependency>

<groupId>org.springframework.data</groupId>

<artifactId>spring-data-releasetrain</artifactId>

<version>Fowler-SR2</version>

<type>pom</type>

<scope>import</scope>

</dependency>

<dependency>

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

<artifactId>spring-boot-dependencies</artifactId>

<version>2.2.2.RELEASE</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</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.

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

#configuring application name

spring.application.name = demoApplication

#configuring port

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. The lines started with* ***#*** *are* ***comments****.*

**Spring Boot Property Categories**

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 table provides 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 start-up. |
| 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 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:

<dependency>

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

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

<version>2.2.2.RELEASE</version>

</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:

<dependency>

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

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

<version>2.0.0.RELEASE</version>

<scope>compile</scope>

</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.

**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:

<dependency>

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

<artifactId>tomcat-embed-core</artifactId>

<version>8.5.23</version>

<scope>compile</scope>

</dependency>

<dependency>

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

<artifactId>tomcat-embed-el</artifactId>

<version>8.5.23</version>

<scope>compile</scope>

</dependency>

<dependency>

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

<artifactId>tomcat-embed-websocket</artifactId>

<version>8.5.23</version>

<scope>compile</scope>

</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.

<dependency>

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

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

<exclusions>

<exclusion>

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

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

</exclusion>

</exclusions>

</dependency>

**Configuring Spring Boot for web-based applications** This application will use additional Java libraries.

* Thymeleaf **Thymeleaf** is a powerful template processing engine for the Spring framework.
* Spring Boot Devtools The **Spring Boot Devtools** automatically recompiles and redeploys the app upon saving and provide

additional development experience enhancements

* Spring Data JPA **Spring Data JPA** makes it easy to implement JPA based repositories and build Spring-powered

applications that use data access technologies.

* H2 **H2** is a Java SQL database. It’s a lightweight database that can be run in-memory.

**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:

public interface EmployeeRepository extends CrudRepository<Employee, Long>

{

Employee findByName(String name);

}

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.

<dependency>

<groupId>org.springframework.data</groupId>

<artifactId>spring-data-jpa</artifactId>

<version>2.2.3.RELEASE</version>

</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).

<dependency>

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

<artifactId>spring-boot-starter-data-jpa</artifactId>

<version>2.2.2.RELEASE</version>

</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.