-

UNIT II

Spring Boot: Creating a Spring Boot Application, Spring Boot Application Annotation, What is Autowiring , Scope of a bean, Logger, Introduction to Spring AOP, Implementing AOP advices, Best Practices : Spring Boot Application

Spring Boot

1: Creating a Spring Boot Application:

How to Create a Spring Boot Project?

Spring Boot is built on the top of the spring and contains all the features of spring. And is becoming a favorite of developers these days because of its rapid production-ready environment which enables the developers to directly focus on the logic instead of struggling with the configuration and setup. Spring Boot is a microservice-based framework and making a production-ready application in it takes very little time.
Following are some of the features of Spring Boot:

- It allows avoiding heavy configuration of XML which is present in spring
- It provides easy maintenance and creation of REST endpoints
- It includes embedded Tomcat-server
- Deployment is very easy, war and jar files can be easily deployed in the tomcat server

For more information please refer to this article: Introduction to Spring Boot
Generally, to develop a Spring Boot Application we choose **Eclipse, Spring Tool Suite**, and **IntelliJ IDEA** IDE. So in this article, we are going to create our spring boot project in these 3 IDEs.

Create a Spring Boot Project in Eclipse IDE

The Eclipse IDE is famous for the Java **Integrated Development Environment (IDE)**, but it has a number of pretty cool IDEs, including the C/C++ IDE, JavaScript/TypeScript IDE, PHP IDE, and more.

Procedure:

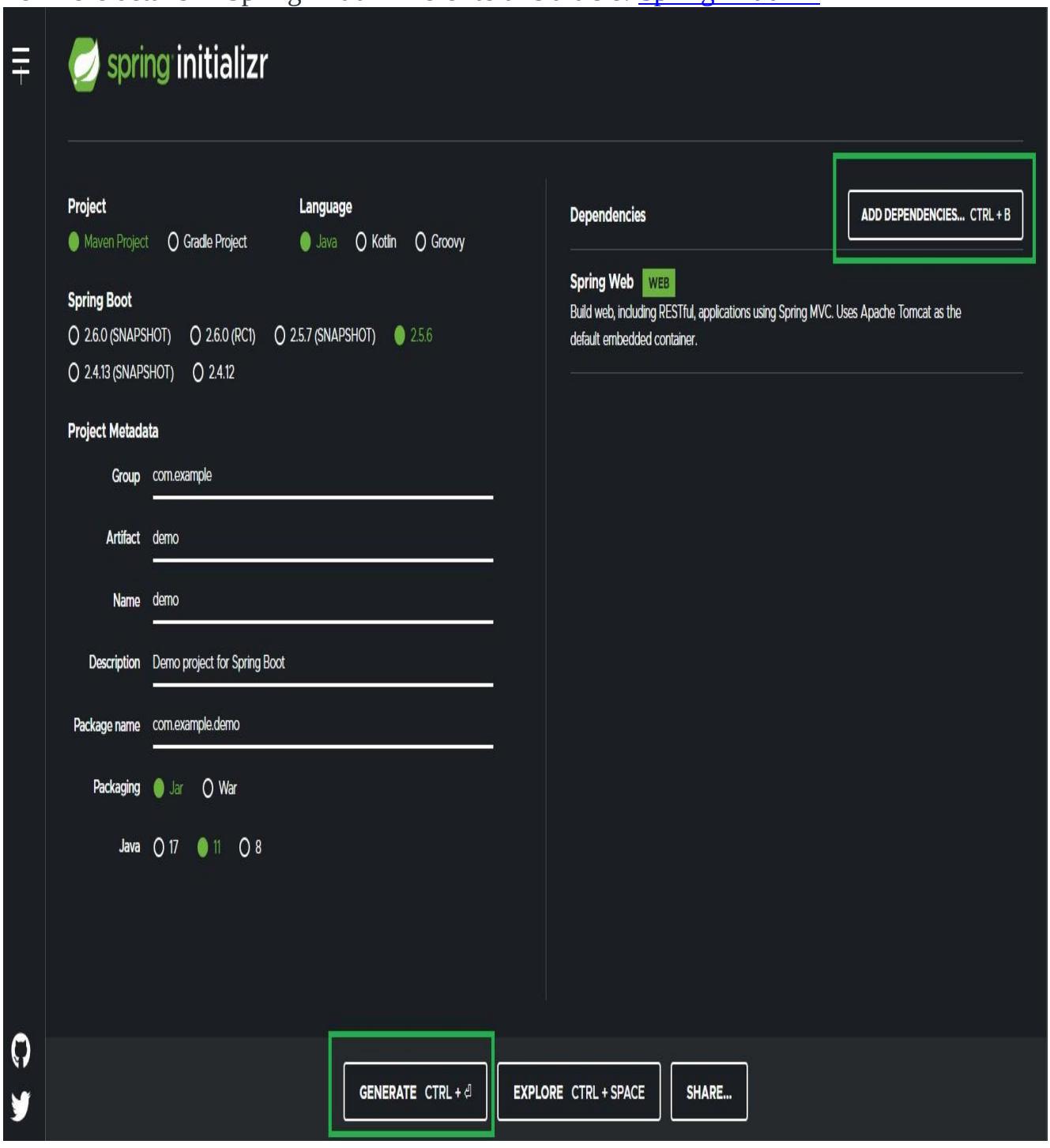
1. Install Eclipse IDE for Enterprise Java and Web Developer
2. Create a Spring Boot Project in Spring Initializr
3. Import Spring Boot Project in Eclipse IDE
4. Search “maven” and choose Existing Maven Project
5. Choose **Next**
6. Click on the **Browse** button and select the extracted zip
7. Click on the **Finish** button and we are done creating the Spring Boot project

Let us discuss these steps in detail alongside visual aids

Step 1: Install Eclipse IDE for Enterprise Java and Web Developer
Please refer to this article How to Install Eclipse IDE for Enterprise Java and Web Development and install the Eclipse IDE.

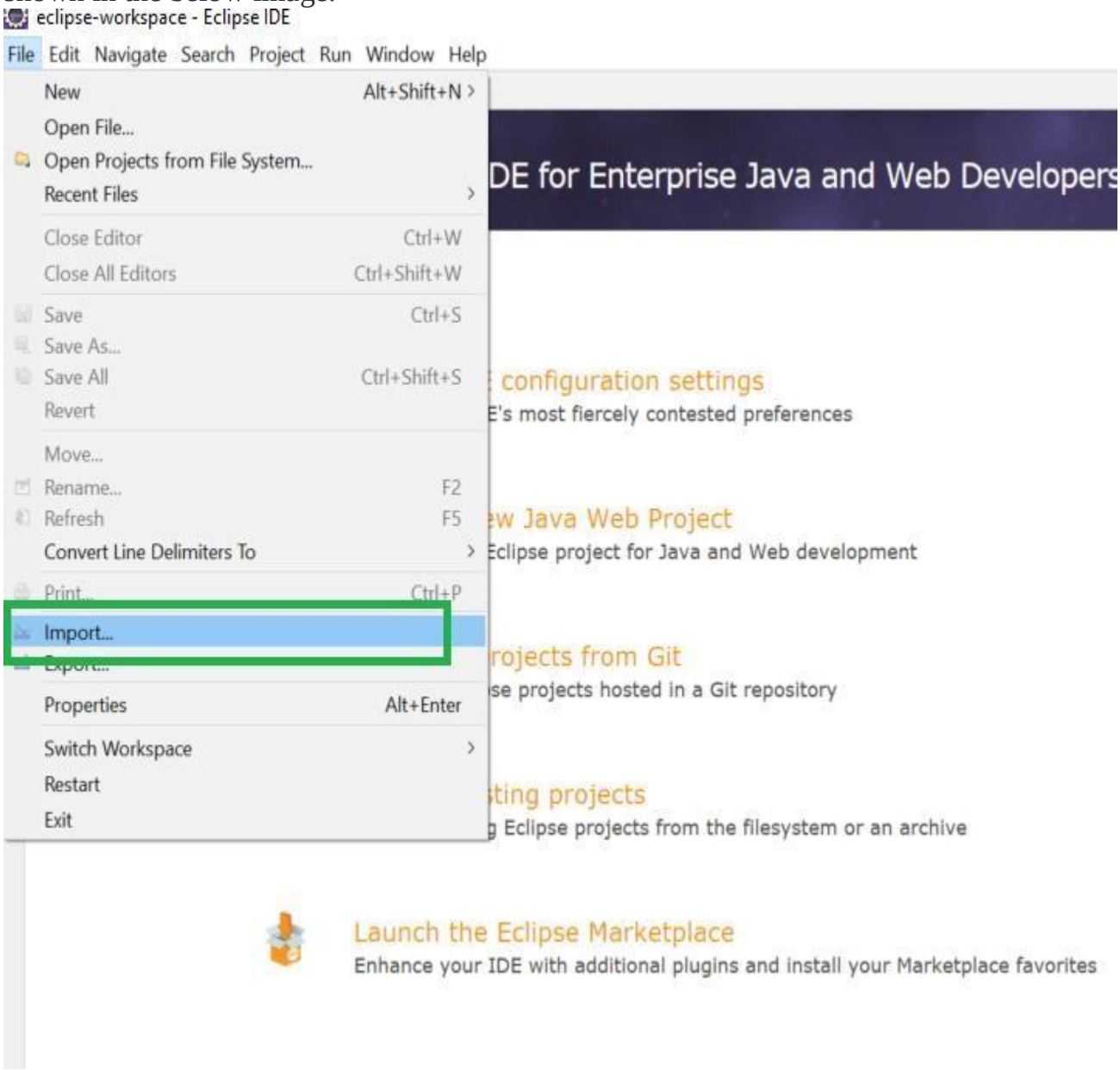
Step 2: Create a Spring Boot Project in Spring Initializr

Go to this link and create a Spring Boot project. Please fill in all the details accordingly and at last click on the **GENERATE** button below. This will download your Spring Boot project in zip format. Now extract the folder into your local machine. For more details in Spring Initializr refer to this article: Spring Initializr

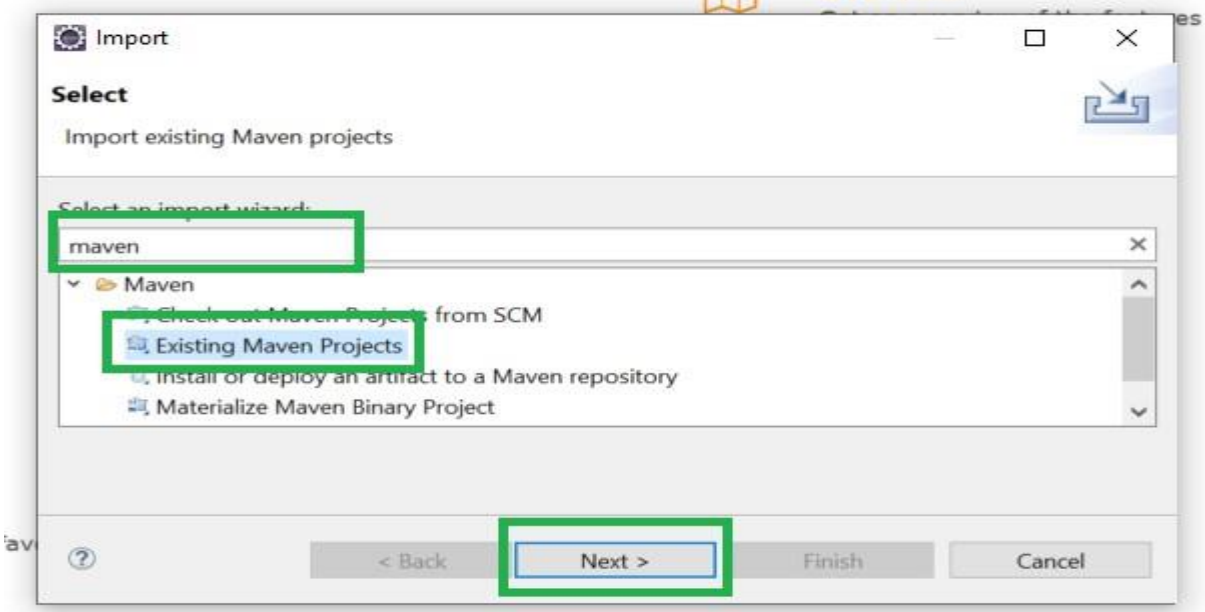


Step 3: Import Spring Boot Project in Eclipse IDE

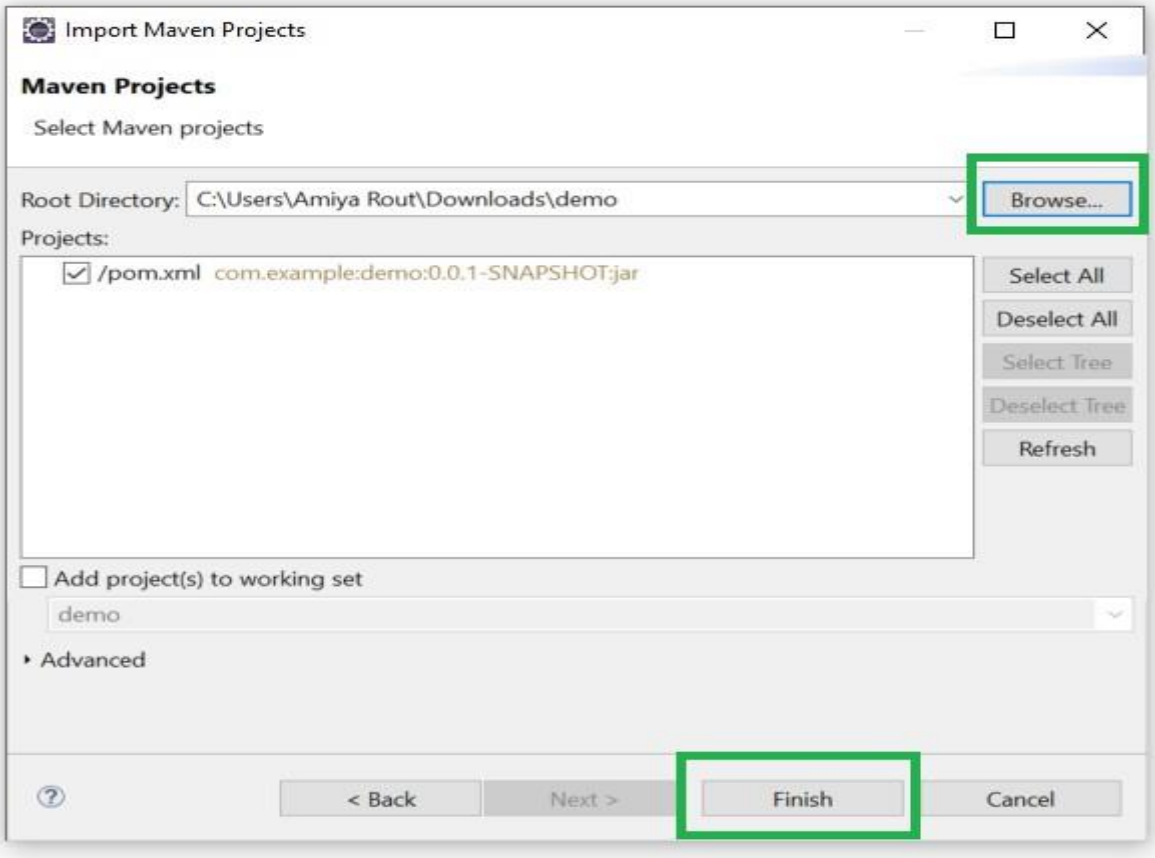
Go to the **Eclipse IDE for Enterprise Java and Web Developer > File > Import** as shown in the below image.



Step 4: Search “maven” and choose **Existing Maven Project** and click on the **Next** button as shown in the below image.

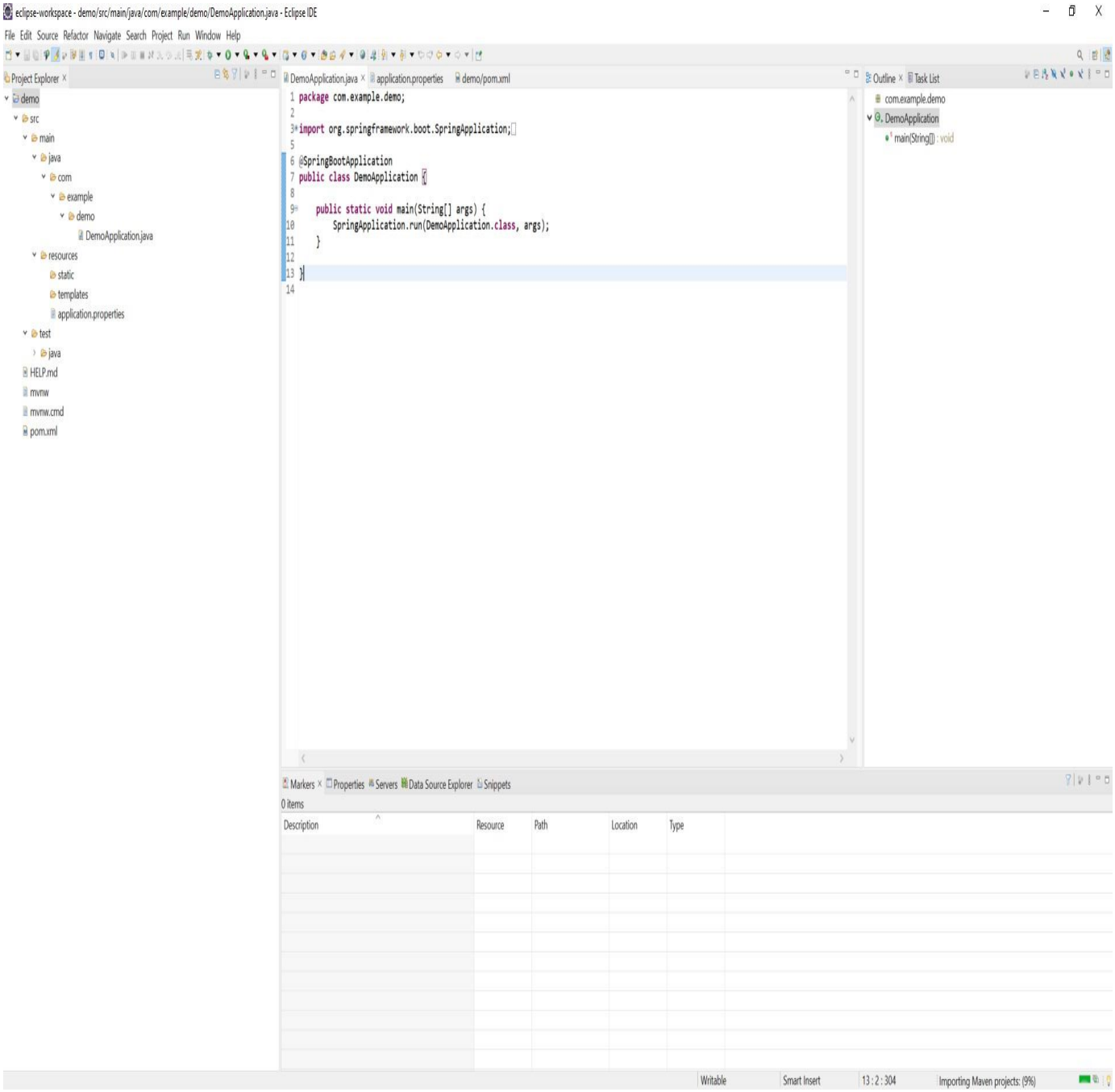


Step 5: Now click on the **Browse** button and select the extracted zip file that has been generated.



Step 6. And at last click on the **Finish** button and we are done creating the Spring Boot project

By now, Spring Boot project has been created as depicted in the below media



Create a Spring Boot Project in STS IDE

Spring Tool Suite (STS) is a java IDE tailored for developing Spring-based enterprise applications. It is easier, faster, and more convenient. And most importantly it is based on Eclipse IDE. STS is free, open-source, and powered by VMware. Spring Tools 4 is the next generation of Spring tooling for the favorite coding environment. Largely rebuilt from scratch, it provides world-class support for developing Spring-based enterprise applications, whether you prefer Eclipse, Visual Studio Code, or Theia IDE.

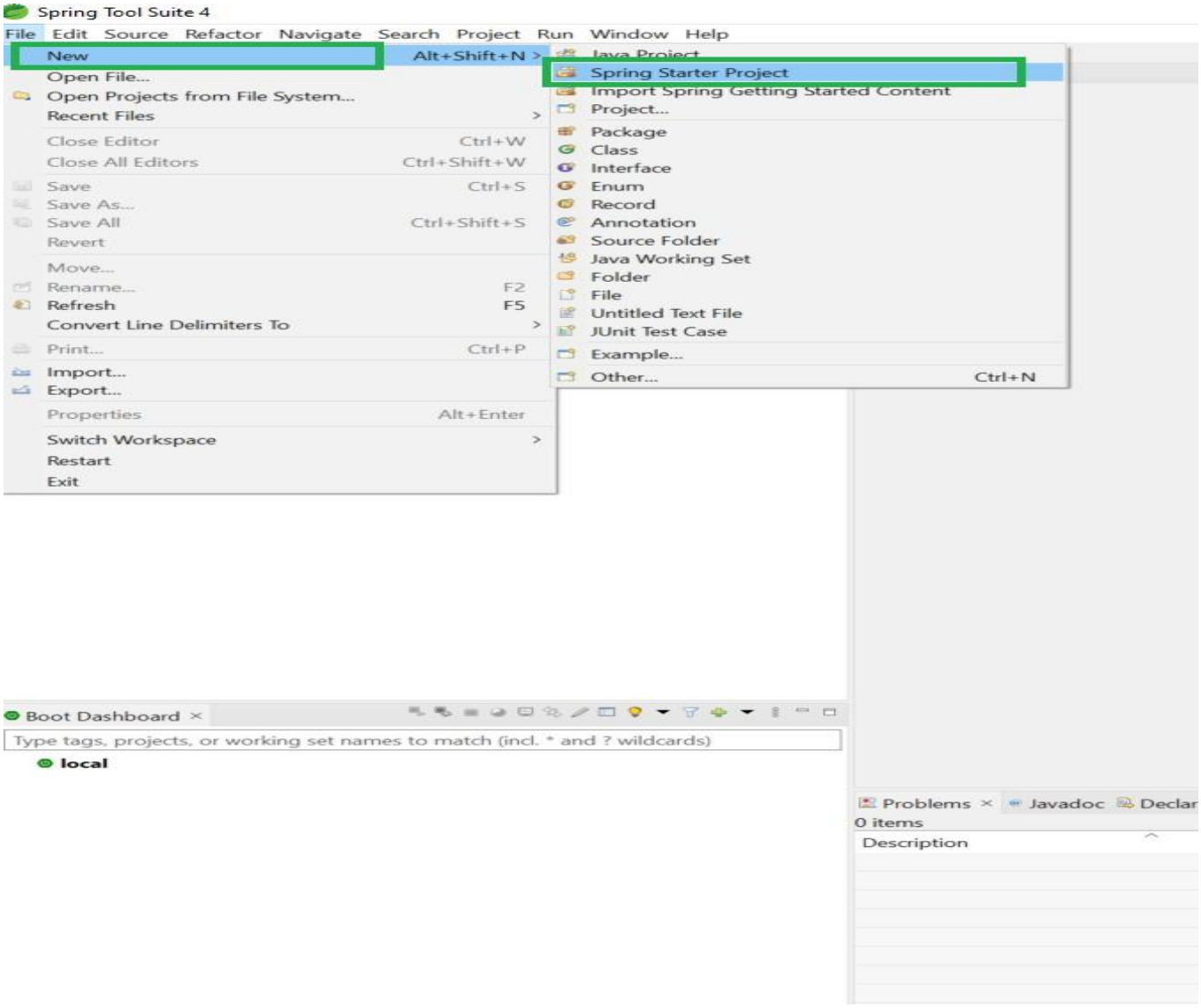
Procedure:

1. Install Spring Tool Suite IDE
2. Create a new Spring project
3. Fill details in the pop-up window and press Next.
4. Choose Spring Boot version and select dependencies and press Next.
5. Click on the 'Finish' button.

Step 1: Install Spring Tool Suite (Spring Tools 4 for Eclipse) IDE

For this user must have pre-requisite knowledge of downloading and installing Spring Tool Suite IDE

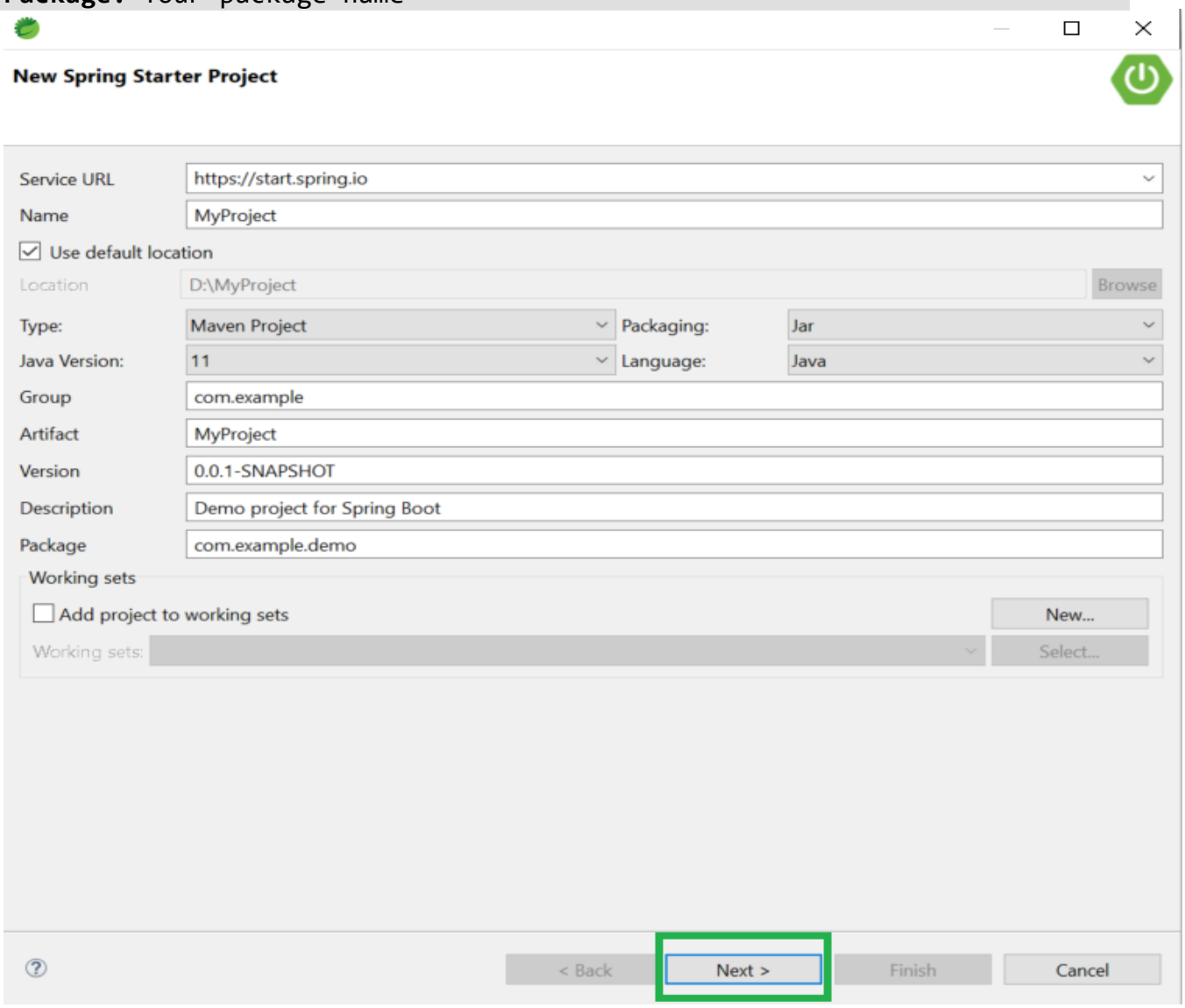
Step 2: Go to the ***File > New > Spring Starter Project*** as shown in the below image as follows:



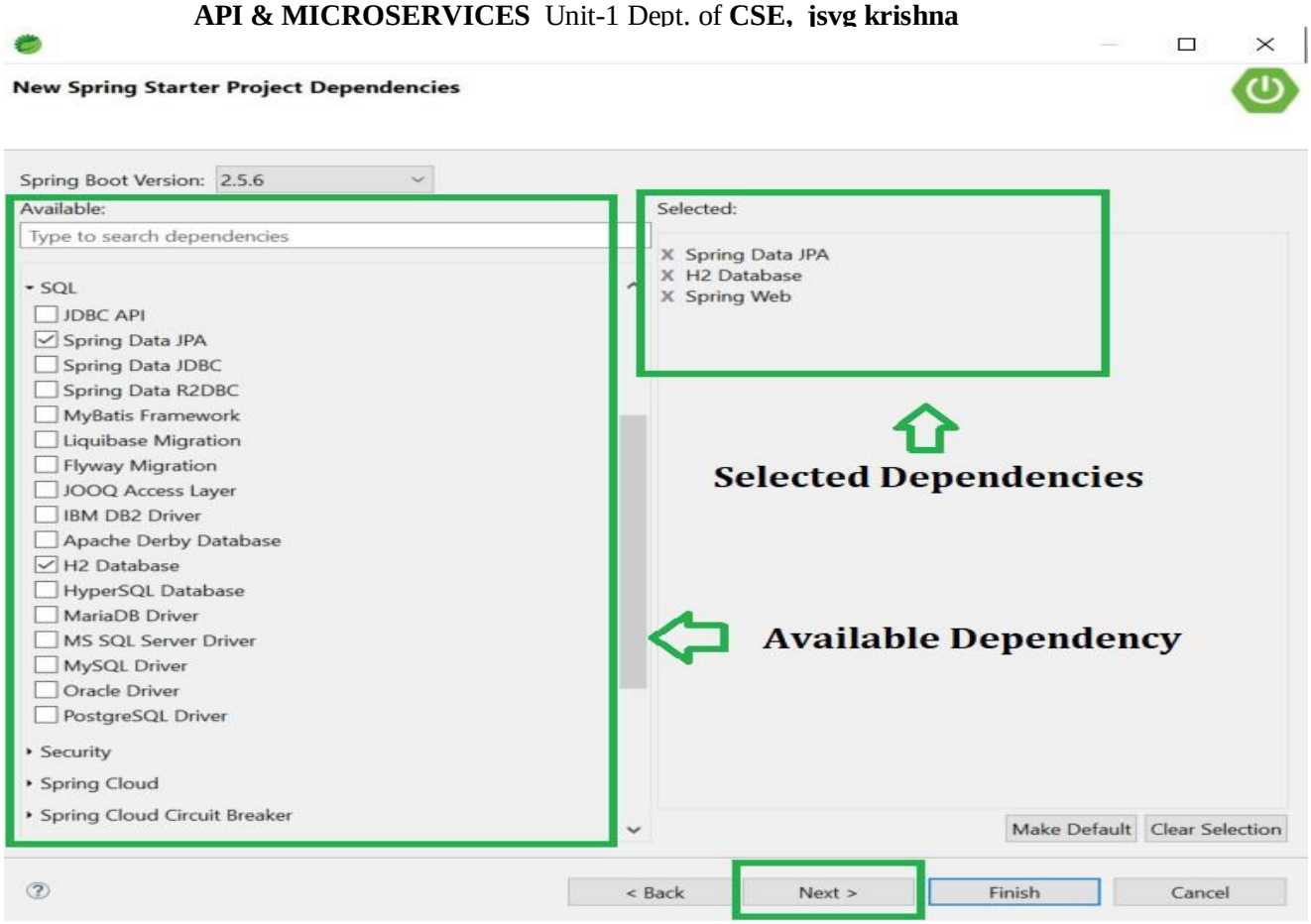
Step 3: In this pop-up window fill the detail below as follows and further click on the **Next** button as shown in the below image.

```
Service URL: Default
Name: Your Project Name
```

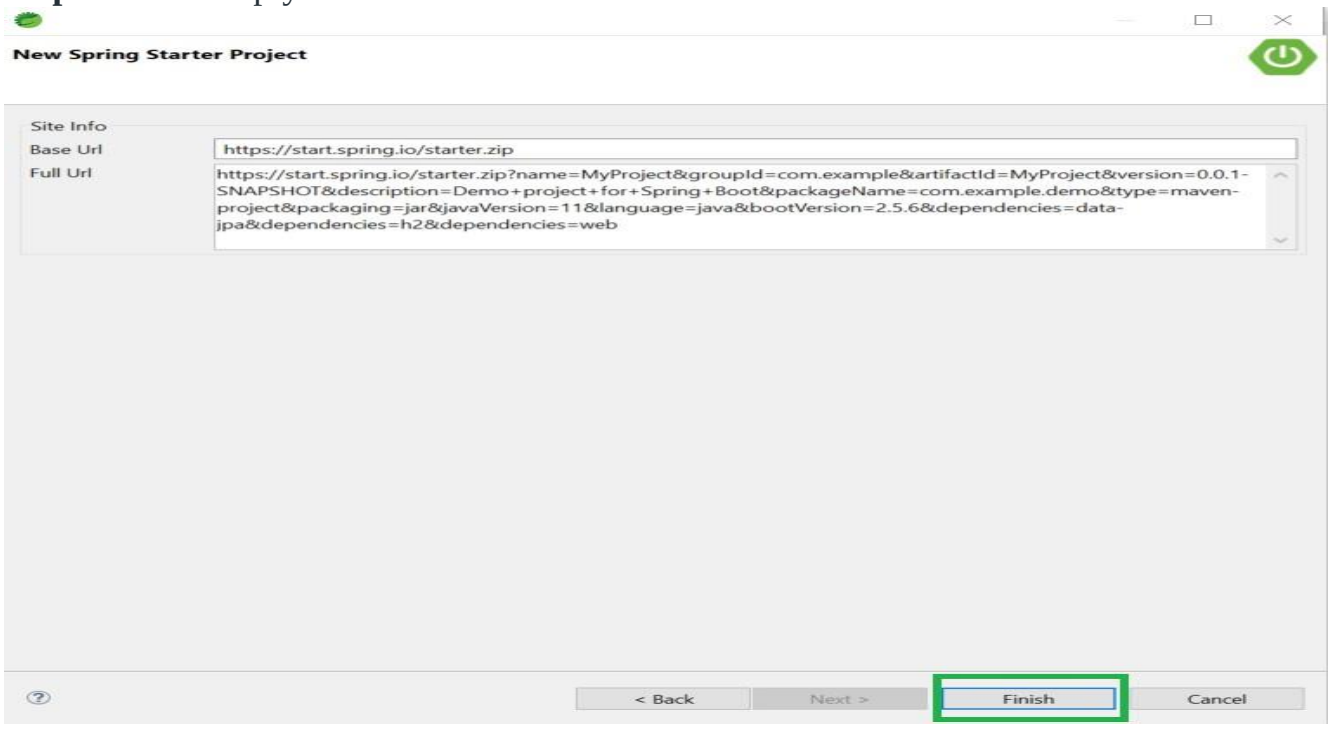
Type: Maven Project
Java Version: 11 or greater than 11
Packaging: As your need
Language: As your need
Group: A unique base name of the company or group that created the project
Artifact: A unique name of the project
Version: Default
Description: As your need
Package: Your package name



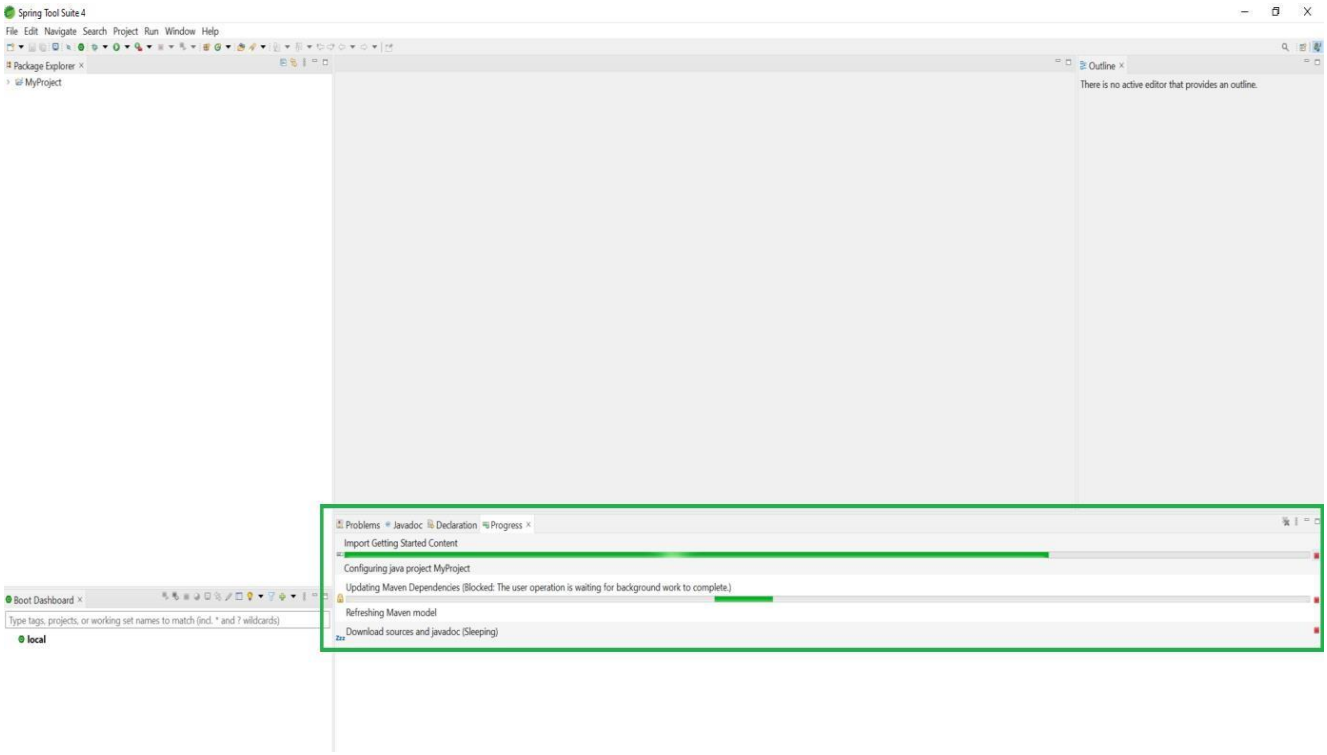
Step 4: Choose your required **Spring Boot Version** and select your dependencies as per your project requirement. And at last click on the **Next** button.



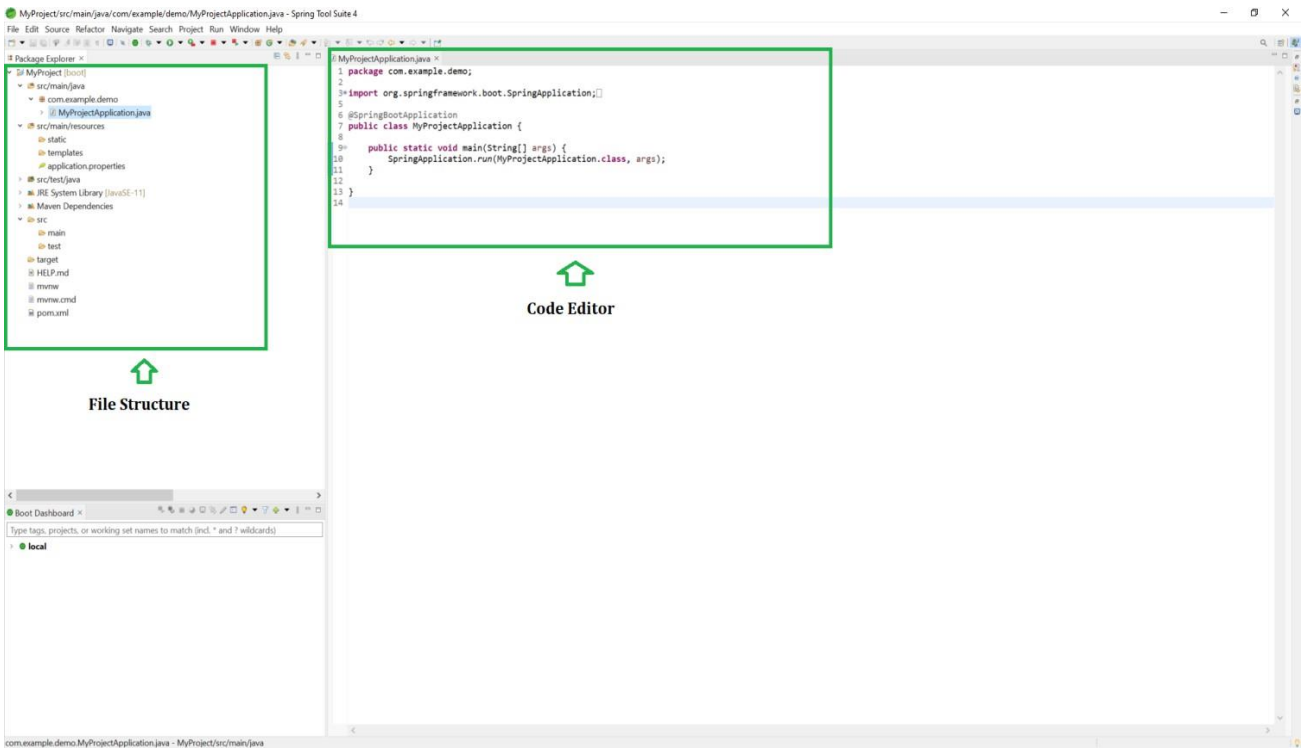
Step 5: Now simply click on the **Finish** button.



Here now, please wait for some time to download all the required files such as dependencies that you have selected in Step4 above.



Below is the Welcome screen after you have successfully Created and Setup Spring Boot Project in Spring Tool Suite



Create a Spring Boot Project in IntelliJ IDEA

IntelliJ is an integrated development environment(IDE) written in Java. It is used for developing computer software. This IDE is developed by Jetbrains and is available as an Apache 2 Licensed community edition and a commercial edition.

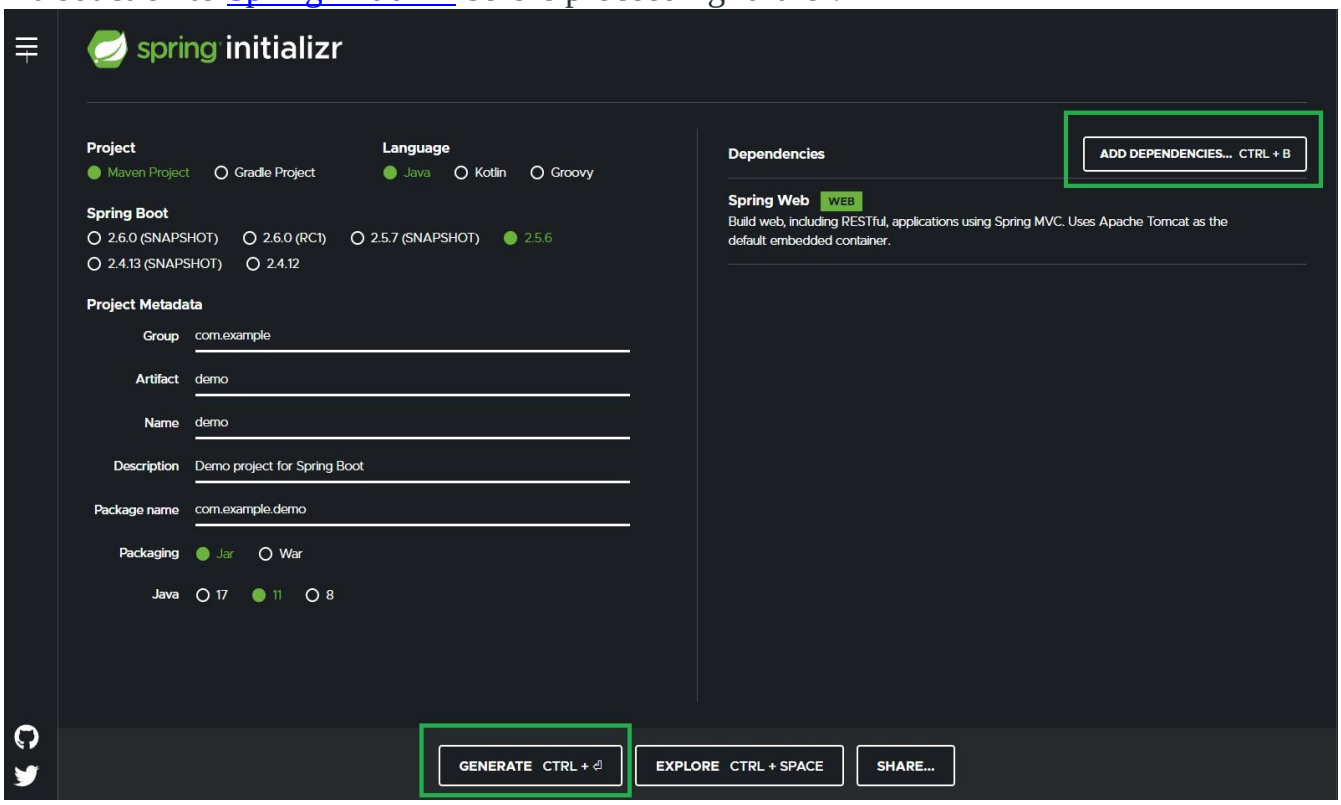
Procedure:

1. Install IntelliJ IDEA on the local machine.
2. Create a Spring Boot Project in Spring Initializr
3. Import Spring Boot Project in IntelliJ IDEA
4. Choose the project that you have created in above step 2.

Step 1: Install IntelliJ IDEA on the local machine for that do go through pre-requisite for installing Intellij Idea on the system.

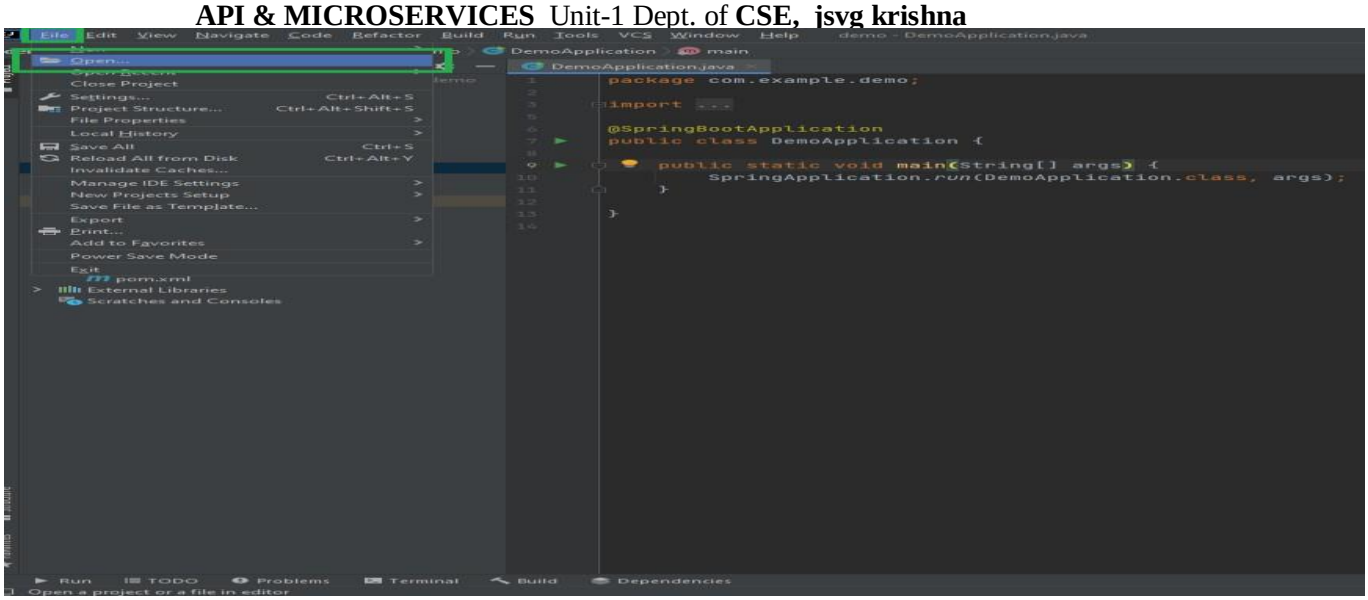
Step 2: Create a Spring Boot Project in Spring Initializr

Create a Spring Boot project and do fill in all the details accordingly and at last click on the **GENERATE** button below. This will download your Spring Boot project in zip format. Now extract the folder into your local machine and do go through the introduction to Spring Initializr before proceeding further.



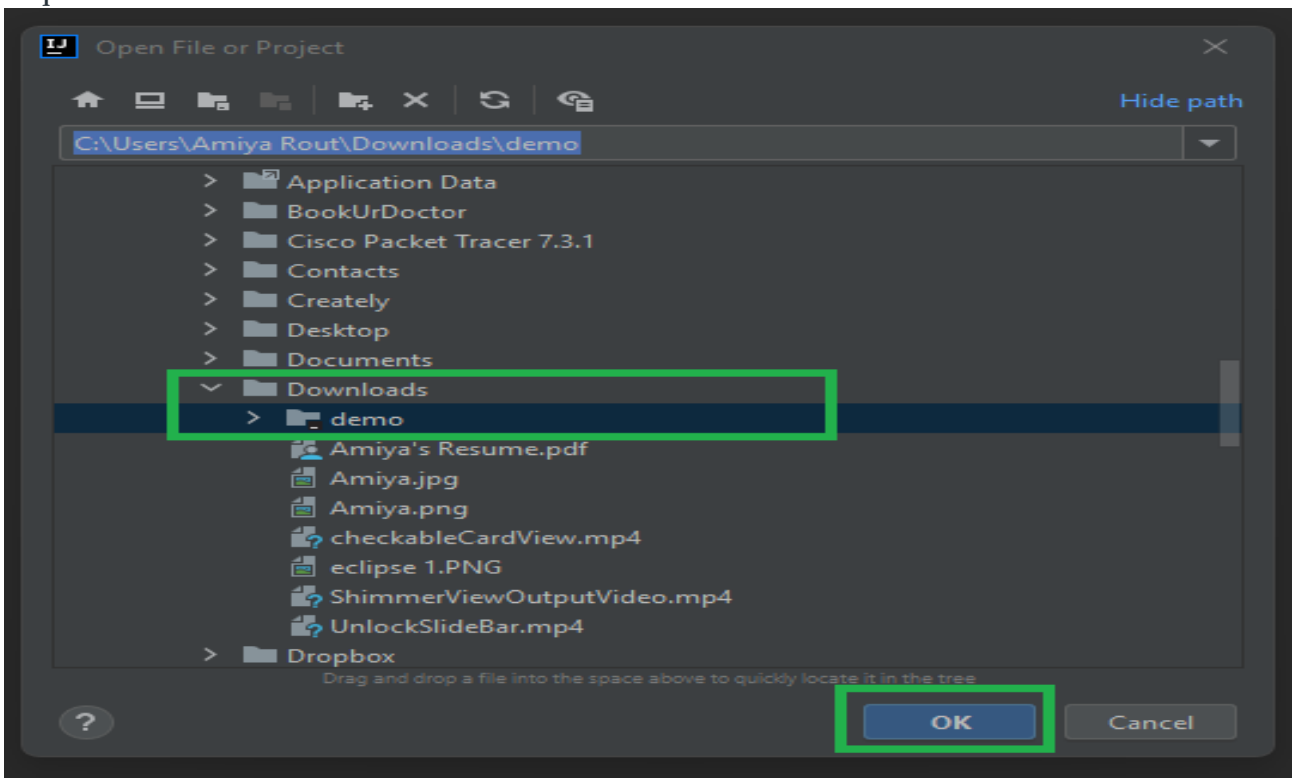
Step 3: Import Spring Boot Project in IntelliJ IDEA

After successfully installing IntelliJ IDEA go to the **File > Open** as seen in the below image.

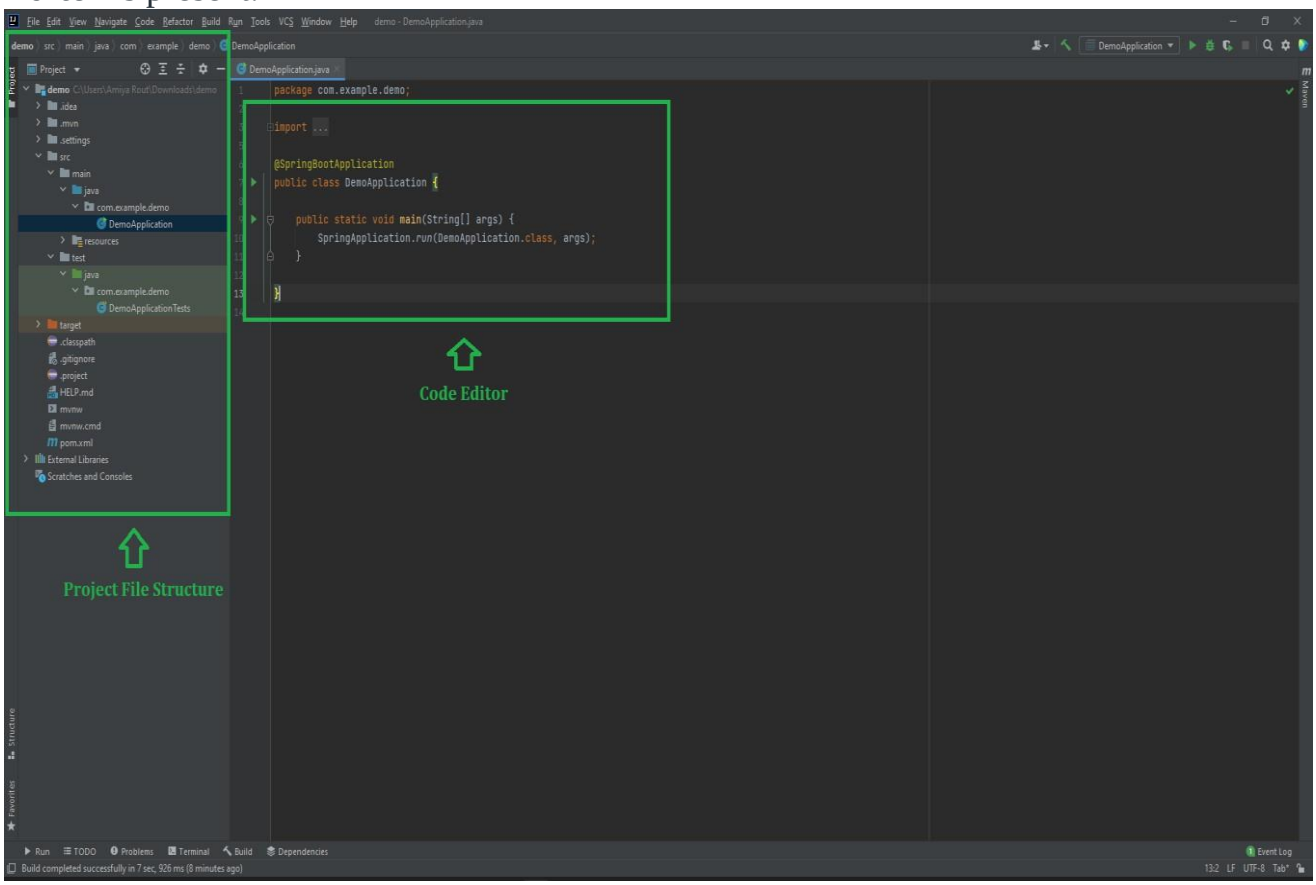


After this, a pop-up window will occur like the following.

Step 4: Here you have to choose the project that you have created in step 2. For example, here we have created the spring boot project named "**demo**" and stored it inside the Downloads folder and you can see we can find the same inside the Downloads folder. At last click on the **OK** button. And we are done creating the Spring Boot project in IntelliJ IDEA. Please wait for some time to download all the required files.



By now, the Spring Boot project has been created as depicted in the below media. The left side contains the **Project file structure** while in the middle you can see the **Code Editor** is present.



2. Spring Boot Application Annotation:

Spring boot is a Java-based framework. It is a backend framework and gained significant popularity in the Java enterprise sector. It enables Java developers to easily and quickly start developing web apps. It is a tool that simplifies the development process using the Spring Framework to create web applications and microservices.

It facilitates programmers to create independent applications that run without the aid of an outside web server. During the initialization phase of your programme, you embed a web server, such as Tomcat, Jetty, or Netty, to complete the construction.

What is @springboot application annotation?

The @springbootapplication annotation is used to create simple spring boot applications. We need to add several annotations to our Application class or Main class to create a spring boot application.

The @SpringBootApplication annotation is a combination of @EnablAutoConfiguration,@Configuration, and @ComponentScan.

The springbootapplication annotation helps the developers to mark configuration classes. The main **three** features of a springbootapplication annotation are as follows:

1. @EnablAutoConfiguration: Because of this feature, the application automatically produces and registers bean methods depending on both the jar files on the included classpath and the beans we define.

Example:

```
@Configuration
@EnableAutoConfiguration

//Application Class or Main Class
public class Main {

   //Main method
   public static void main(String[] args) {
      SpringApplication.run(Main.class, args);
   }
}
```

2. @Configuration: It tries to configure your Spring application automatically based on the jar dependencies you provided. You can use properties files, YAML files, environment variables, and command-line arguments to externalise configuration.

Example:

```
@Configuration

//Application Class or Main Class
public class Main {

   //Main method

   public static void main(String[] args) {
      SpringApplication.run(Main.class, args);
   }



   @bean
   // bean method's name or id
         public CN cnBean() {          // Returning the CN object

      return new CN();
   }
```

3. @ComponentScan: When searching for components in Spring, the @ComponentScan annotation is used to define packages.

Example:

```
@Configuration
@ComponentScan

//Application Class or Main Class
```

```
public class Main{

 //Main method
   public static void main(String[] args) {
       SpringApplication.run(Main.class, args);
    }
}
```

Spring Boot SpringApplication Class

The Spring Boot SpringApplication class is a central component of the Spring Boot framework. It is used to bootstrap and launch a Spring application from a Java main method.

The SpringApplication class automatically performs a number of tasks, including:

- Creating an ApplicationContext instance
- Scanning for configuration classes
- Registering command line arguments as Spring properties
- Refreshing the ApplicationContext
- Triggering any CommandLineRunner beans

To use the SpringApplication class, we need to create an instance of it and call the **run()** method. The run() method takes a list of arguments as input, which is used to configure the application.

The following code example shows how to bootstrap and launch a simple Spring Boot application:

```
public class MyApplication {
  public static void main(String[] args) {
     SpringApplication.run(MyApplication.class, args);
  }
}
```

How to Implement @springboot application annotation?

An application or main class with the @SpringBootApplication annotation is required to run a Spring Boot application. The syntax is as follows:

```
package example;


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


@SpringBootApplication
// works same as @EnableAutoConfiguration, @Configuration, and @ComponentScan.
```

```
public class Application {


   public static void main(String[] args) {
       SpringApplication.run(Application.class, args);
    }
}
```

The above code simply calls the SpringApplication.run() method. This launches the embedded beans and launches the Spring application as separate applications. One can place this code in the root package. It will help in component scanning and searching for beans in every sub-package.

Optional Parameters:

The @SpringBootApplication annotation accepts the following parameters:

- **Class<?>[] exclude:** Make certain auto-configuration classes irrelevant by excluding them.
- **Class<?>[] scanBasePackageClass:** A type-safe alternative to scanBasePackages() for providing the packages to search for annotated components is scanBasePackageClass.
- **String[] excludeName:** This parameter excludes the application of a particular auto-configuration class name.
- **String[] scanBasePackages:** This parameter is used to search for annotated components using scanBasePackages.

Optional Features:

The @SpringBootApplication annotation offers aliases to modify the @EnableAutoConfiguration @Configuration and @ComponentScan's attributes. We can replace the springbootapplication annotation with any of these annotations, but none of these features is required. For example, you can use the springbootapplication annotation without the @ComponentScan method.

Here is the code for implementing the springbootapplication annotation without the @ComponentScan method.

```
package example;
import org.springframework.boot.SpringApplication;

import org.springframework.context.annotation.Import;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.ComponentScan;



@Configuration
@EnableAutoConfiguration
//Not using the @ComponentScan
```

```
@Import({ FirstConfig.class, AnotherConfig.class })
//User-defined annotations are imported explicitly
public class Main{
   public static void main(String[] args) {
      SpringApplication.run(Main.class, args);
   }
}
```

Usage and Benefits:

- The @SpringBootApplication annotation is typically used to annotate the main class of a Spring Boot application. It provides a convenient way to enable Java-based Spring configuration, component scanning, and Spring Boot's auto-configuration feature.
- By using this annotation, developers can avoid writing boilerplate code for configuring the Spring application context, specifying the web server, defining properties, and setting up various beans. This simplifies the setup process and allows for a more streamlined development experience.

3. What is Autowiring:

Autowiring in the Spring framework can inject dependencies automatically. The Spring container detects those dependencies specified in the configuration file and the relationship between the beans. This is referred to as **Autowiring in Spring**. To enable Autowiring in the Spring application we should use @Autowired annotation. Autowiring in Spring internally uses constructor injection. An autowired application requires fewer lines of code comparatively but at the same time, it provides very little flexibility to the programmer.

Modes of Autowiring:

Modes	Description
No	This mode tells the framework that autowiring is not supposed to be done. It is the default mode used by Spring.
byName	It uses the name of the bean for injecting dependencies.

Modes	Description
byType	It injects the dependency according to the type of bean.
Constructor	It injects the required dependencies by invoking the constructor.
Autodetect	The autodetect mode uses two other modes for autowiring – constructor and byType.

1. No:

This mode tells the framework that autowiring is not supposed to be done. It is the default mode used by Spring.

```
<bean id="state" class="sample.State">
 <property name="name" value="UP" />
</bean>
<bean id="city" class="sample.City"></bean>
```

2. byname:

It uses the name of the bean for injecting dependencies. However, it requires that the name of the property and bean must be the same. It invokes the setter method internally for autowiring.

```
<bean id="state" class="sample.State">
 <property name="name" value="UP" />
</bean>
<bean id="city" class="sample.City" autowire="byName"></bean>
```

3. byType:

It injects the dependency according to the type of the bean. It looks up in the configuration file for the class type of the property. If it finds a bean that matches, it injects the property. If not, the program throws an error. The names of the property and bean can be different in this case. It invokes the setter method internally for autowiring.

```
<bean id="state" class="sample.State">
 <property name="name" value="UP" />
</bean>
<bean id="city" class="sample.City" autowire="byType"></bean>
```

4. constructor:

It injects the required dependencies by invoking the constructor. It works similar to the “byType” mode but it looks for the class type of the constructor arguments. If none or more than one bean are detected, then it throws an error, otherwise, it autowires the “byType” on all constructor arguments.

```
<bean id="state" class="sample.State">
 <property name="name" value="UP" />
</bean>
<bean id="city" class="sample.City" autowire="constructor"></bean>
```

5. autodetect:

The autodetect mode uses two other modes for autowiring – constructor and byType. It first tries to autowire via the constructor mode and if it fails, it uses the byType mode for autowiring. It works in Spring 2.0 and 2.5 but is deprecated from Spring 3.0 onwards.

```
<bean id="state" class="sample.State">
 <property name="name" value="UP" />
</bean>
<bean id="city" class="sample.City" autowire="autodetect"></bean>
```

Example of Autowiring:

Sample Program for the byType Mode:

State.java

- Java

```
public class State {
    private String name;
    public String getName() { return name; }
    public void setName(String s) { this.name = s; }
}
```

City.java

- Java

```
class City {
    private int id;
    private String name;
    private State s;
    public int getID() { return id; }
    public void setId(int eid) { this.id = eid; }
    public String getName() { return name; }
    public void setName(String st) { this.name = st; }
    public State getState() { return s; }
```

```
    @Autowired public void setState(State sta)
    {
        this.s = sta;
    }
    public void showCityDetails()
    {
        System.out.println("City Id : " + id);
        System.out.println("City Name : " + name);
        System.out.println("State : " + s.getName());
    }
}
```

Spring bean configuration file:

- XML

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:context="http://www.springframework.org/schema/context"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context.xsd">
<bean id="state" class="sample.State">
<property name="name" value="UP" />
</bean>
<bean id="city" class="sample.City" autowire="byName"></bean>
</beans>
```

Test Program file: DemoApplication.java

- Java

```
@SpringBootApplication
public class DemoApplication {
    public static void main(String[] args)
    {
        SpringApplication.run(DemoApplication.class,  args);
        ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
        City cty = context.getBean("city", City.class);
        cty.setId(01);
        cty.setName("Varanasi");
        State st = context.getBean("state", State.class);
        st.setName("UP");
```

```
        cty.setState(st);
        cty.showCityDetails();
    }
}
```

Output:

```
City ID : 01
City Name : Varanasi
State : UP
```

4. Scope of a bean:

n Spring, a bean's scope defines the lifecycle and visibility of the bean instance within the application context. It determines how long the bean remains in memory and how it is accessed by different parts of the application. Spring offers several built-in bean scopes, each serving specific use cases and scenarios.

1. Singleton Scope: One Bean to Rule Them All

- The Singleton scope is the default scope for Spring Beans if no explicit scope is defined.
- When a bean is defined with the Singleton scope, the Spring IoC container creates and manages a single instance of the bean for the entire application context.
- Every time the bean is requested, the same instance is returned, and subsequent requests for the bean will reuse this single instance.
- This means that all parts of the application that use the bean will share the same instance, potentially leading to thread-safety concerns.
- Singleton beans are generally used for stateless services and managing shared resources across the application.

Example:

In this example, we have a Spring Bean named `SingletonBean` with a Singleton scope. This means that Spring will only create a single instance of this bean for the entire application context.

When we access the SingletonBean using the Spring context, we always get the same instance, and any changes made to that instance are visible across all the references. In the given code snippet, the SingletonBean contains a counter that increments each time we call the `getCounter()` method.

```
import org.springframework.stereotype.Component;
import org.springframework.beans.factory.config.ConfigurableBeanFactory;
```

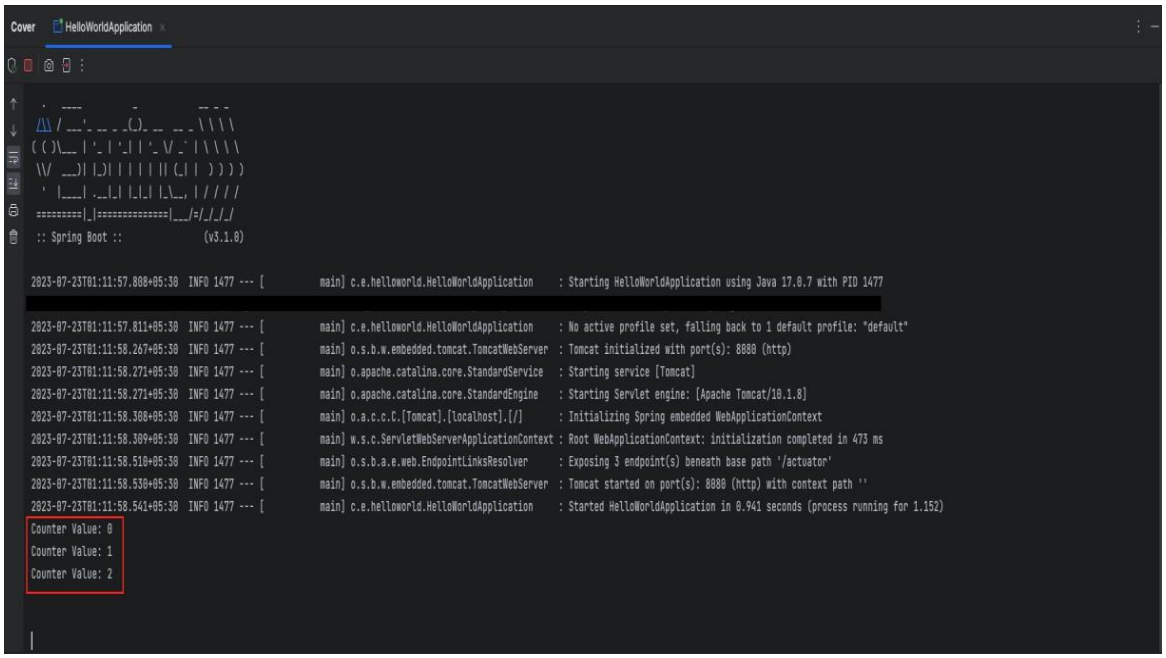
```
// Explicitly stating that a particular bean is of type singleton. Two ways to
define the scope.
// @Scope(value = ConfigurableBeanFactory.SCOPE_SINGLETON)
// @Scope("Singleton")
@Component
public class SingletonBean {
    private int counter = 0;

    public int getCounter() {
        return counter++;
    }
}
```

SingletonBean.java

By default, each and every bean is of type Singleton but you can explicitly specify that a particular bean is of type single using a Scope annotation with value as a singleton

```
@SpringBootApplication
public class HelloWorldApplication {

    public static void main(String[] args) {
        ConfigurableApplicationContext applicationContext =
SpringApplication.run(HelloWorldApplication.class,  args);

        SingletonBean bean1 = applicationContext.getBean(SingletonBean.class);
        System.out.println("Counter Value: " + bean1.getCounter()); // Output: 0

        SingletonBean bean2 = applicationContext.getBean(SingletonBean.class);
        System.out.println("Counter Value: " + bean2.getCounter()); // Output: 1

        SingletonBean bean3 = applicationContext.getBean(SingletonBean.class);
        System.out.println("Counter Value: " + bean3.getCounter()); // Output: 2
    }
```

Output:



This demonstrates that each time we access the `getCounter()` method, the same SingletonBean instance is used, and the counter increments accordingly.

2. Prototype Scope: Every Bean for Itself

- The Prototype scope defines that a new instance of the bean should be created every time it is requested from the Spring IoC container.
- Unlike Singleton, Prototype beans do not maintain a single shared instance; instead, each request for the bean creates a new instance.
- This makes Prototype beans ideal for stateful components as each client requesting the bean gets a fresh instance.
- Prototype-scoped beans can be more memory-intensive, so use them carefully.

Example:

In this example, we have a Spring Bean named `PrototypeBean` with a Prototype scope. This means that every time we request a bean of this type from the Spring context, a new instance is created.

When we access the PrototypeBean using the Spring context, we get a new instance of the bean each time, and any changes made to that instance are isolated to that particular instance only. In the given code snippet, the PrototypeBean contains a counter that increments each time we call the `getCounter()` method.

```
import org.springframework.stereotype.Component;
import org.springframework.beans.factory.config.ConfigurableBeanFactory;

@Component
@Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE)
class PrototypeBean {
    private int counter = 0;

    public int getCounter() {
        return counter++;
    }

}
```

```
@SpringBootApplication
public class HelloWorldApplication {

    public static void main(String[] args) {
        ConfigurableApplicationContext applicationContext =
SpringApplication.run(HelloWorldApplication.class,  args);

        PrototypeBean bean1 = applicationContext.getBean(PrototypeBean.class);
        System.out.println("Counter Value: " + bean1.getCounter()); // Output: 0
```

```
        PrototypeBean bean2 = applicationContext.getBean(PrototypeBean.class);
        System.out.println("Counter Value: " + bean2.getCounter()); // Output: 0

        PrototypeBean bean3 = applicationContext.getBean(PrototypeBean.class);
        System.out.println("Counter Value: " + bean3.getCounter()); // Output: 0
    }


}
```

Output:

This demonstrates that each time we access the `getCounter()` method, a new PrototypeBean instance is created, and the counter starts from zero for each instance independently.

3. Request Scope: Unique Beans for Each HTTP Request

- The Request scope is specific to web applications and only valid in the context of a web-aware Spring ApplicationContext.
- When a bean is defined with the Request scope, a new instance of the bean is created for each HTTP request.
- This is particularly useful for handling request-specific data, as each user's request gets its own isolated instance of the bean.

- This scope ensures that each user request is isolated and doesn't interfere with other requests.
- Request-scoped beans are thread-safe within the context of a single HTTP request.

The Working Principle

In a web application context, when a user sends an HTTP request to the server, Spring creates a new instance of the bean associated with the Request scope. This instance remains available throughout the duration of that specific request. Once the request is processed and the response is sent back to the user, the bean is discarded, and its resources are released.

Example:
In this example, we have a special bean (`RequestBean`) that keeps track of the number of times it's accessed within a single web request. The REST Controller (`HelloWorldController`) uses this bean to show how the count resets with each new HTTP request, ensuring a fresh count for every user interaction.

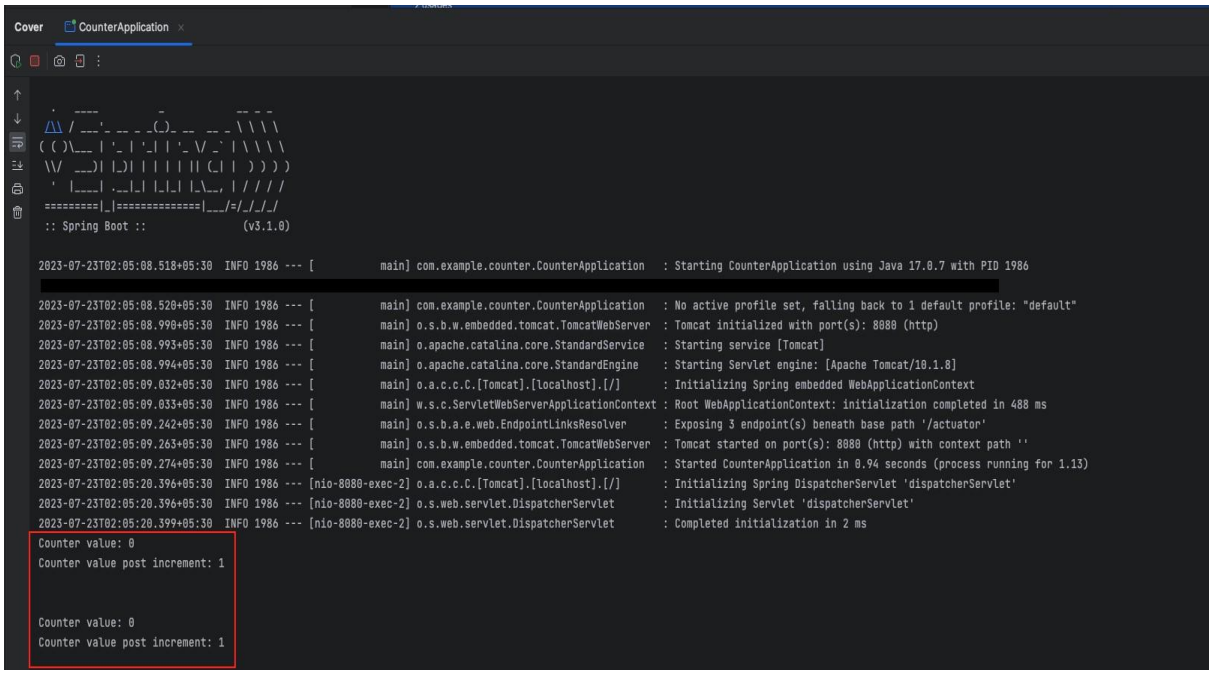
```
import org.springframework.context.annotation.Scope;
import org.springframework.context.annotation.ScopedProxyMode;
import org.springframework.stereotype.Component;

@Component
@Scope(value = "request", proxyMode = ScopedProxyMode.TARGET_CLASS)
public class RequestBean {
    private int counter = 0;

    public int getCounter() {
        return counter++;
    }

}
```

By using `proxyMode = ScopedProxyMode.TARGET_CLASS`, we are resolving the issue of accessing request-scoped beans from singleton-scoped beans in a web context. It ensures that each request gets its own instance of the `RequestBean`, and the proxy takes care of managing the correct instances based on the request context.

```
@RestController
public class CounterController {

    @Autowired
    private RequestBean requestBean;

    @GetMapping("/get-counter-value")
    public String counterValue() {
```

```
        int currentCounterValue = requestBean.getCounter();
        System.out.println("Counter value: " + currentCounterValue);

        currentCounterValue = requestBean.getCounter();
        System.out.println("Counter value post increment: " +
currentCounterValue);

        // return incremented counter value
        return "Counter value: " + currentCounterValue;
    }
}
```

Output:



Each time you access the `/get-counter-value` endpoint, a new instance of `RequestBean` is created for that specific request, and the counter increments independently. This demonstrates the request scope behavior, and the proxy ensures that each request gets its own instance of the `RequestBean`.

4. Session Scope: Beans Bound to User Sessions

- The Session scope is also specific to web applications and valid in the context of a web-aware Spring ApplicationContext.
- When a bean is defined with the Session scope, a new instance of the bean is created for each user session.
- This ensures that each user interacting with the application gets a separate instance of the bean, maintaining session-specific data.
- Session-scoped beans are useful for managing user-specific information throughout their session.

The Working Principle

In a web application context, when a user starts a session (e.g., by logging in), Spring creates a new instance of the bean associated with the Session scope. This bean instance remains available and shared throughout the entire duration of that user's session. Once the user's session ends (e.g., by logging out or session timeout), the bean is discarded, and its resources are released

Example:
In this example, we have a Spring Bean named `SessionBean`, designed with session scope, which means it retains its state throughout a user's session. The `SessionBean` has a counter that increments each time the `getCounter()` method is called. We also have a REST Controller named `CounterController`, which utilizes the `SessionBean`. When the endpoint `/get-counter-value` is accessed, the `CounterController` calls the `getCounter()` method multiple times. As a result, the counter increments with each call within the same user session, and the updated value is returned in the response. This session-scoped bean ensures that each user maintains an independent counter value, allowing separate counting for different users interacting with the application.

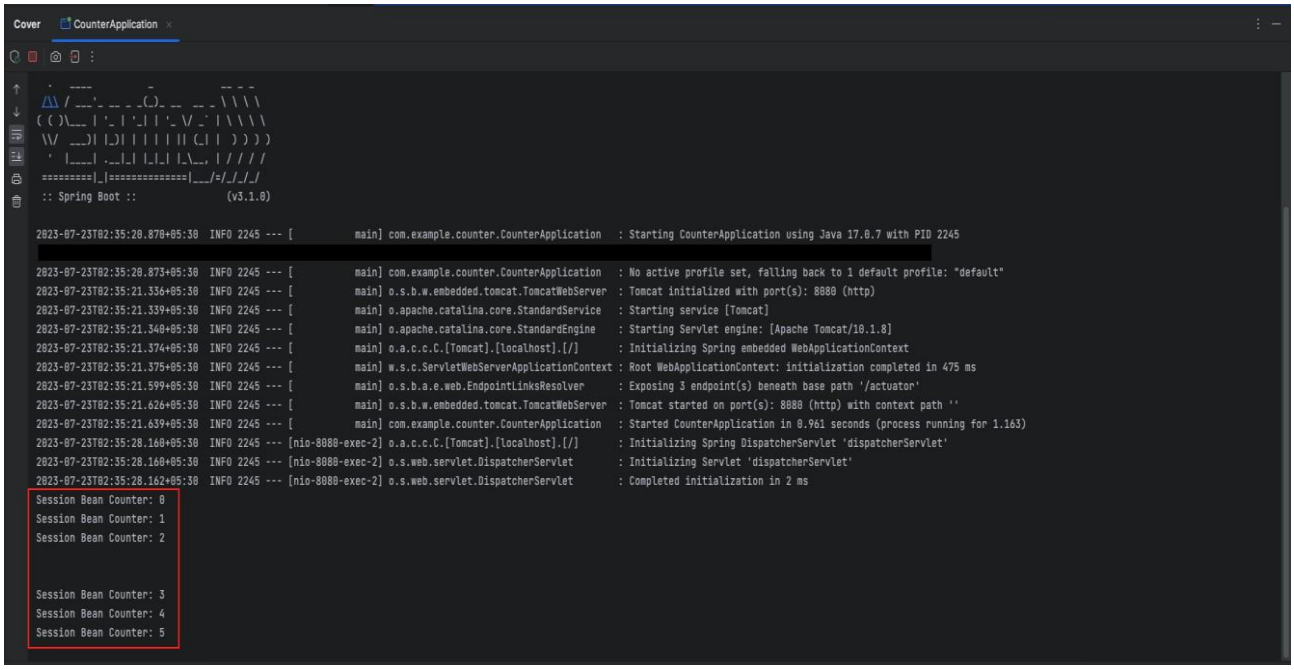
```
mport org.springframework.context.annotation.Scope;
import org.springframework.context.annotation.ScopedProxyMode;
import org.springframework.stereotype.Component;

@Component
@Scope(value = "session", proxyMode = ScopedProxyMode.TARGET_CLASS)
public class SessionBean {
    private int counter = 0;

    public int getCounter() {
        return counter++;
    }
}
```

By using `proxyMode = ScopedProxyMode.TARGET_CLASS`, we are resolving the issue of accessing session-scoped beans from singleton-scoped beans in a web context. It ensures that each user session gets its own unique instance of the `SessionBean`, and the proxy handles managing the correct instances based on the user's session context.

```
@RestController
public class CounterController {

    @Autowired
    private SessionBean sessionBean;

    @GetMapping("/get-counter-value")
    public String counterValue() {
        int couterValue = sessionBean.getCounter();
        System.out.println("Session Bean Counter: " + couterValue);

        couterValue = sessionBean.getCounter();
```

```
        System.out.println("Session Bean Counter: " + couterValue);

        couterValue = sessionBean.getCounter();
        System.out.println("Session Bean Counter: " + couterValue);

        // return incremented counter value
        return "Counter value: " + couterValue;
    }
}
```

Output:



With the `SessionBean` set to session scope, every time a user starts a new session and accesses the `/get-counter-value` endpoint, a unique instance of SessionBean is created. The counter within the SessionBean increments independently for each method call within the same user session. This demonstrates the session scope behavior, where the same bean instance persists throughout the user's session, ensuring separate counting for each user's interaction with the application.

5. Application Scope: Beans Across the Entire Application

- The Application scope is specific to web applications and valid in the context of a web-aware Spring ApplicationContext.
- When a bean is defined with the Application scope, a single instance of the bean is created for the entire ServletContext.
- This means that the same instance will be shared across all user sessions and requests within the application.
- Application-scoped beans are useful for objects that need to maintain state globally across the application.

The Working Principle

In a web application context, when the application starts, Spring creates a single instance of the bean associated with the Application scope. This instance remains available throughout the entire lifecycle of the web application until it is shut down.

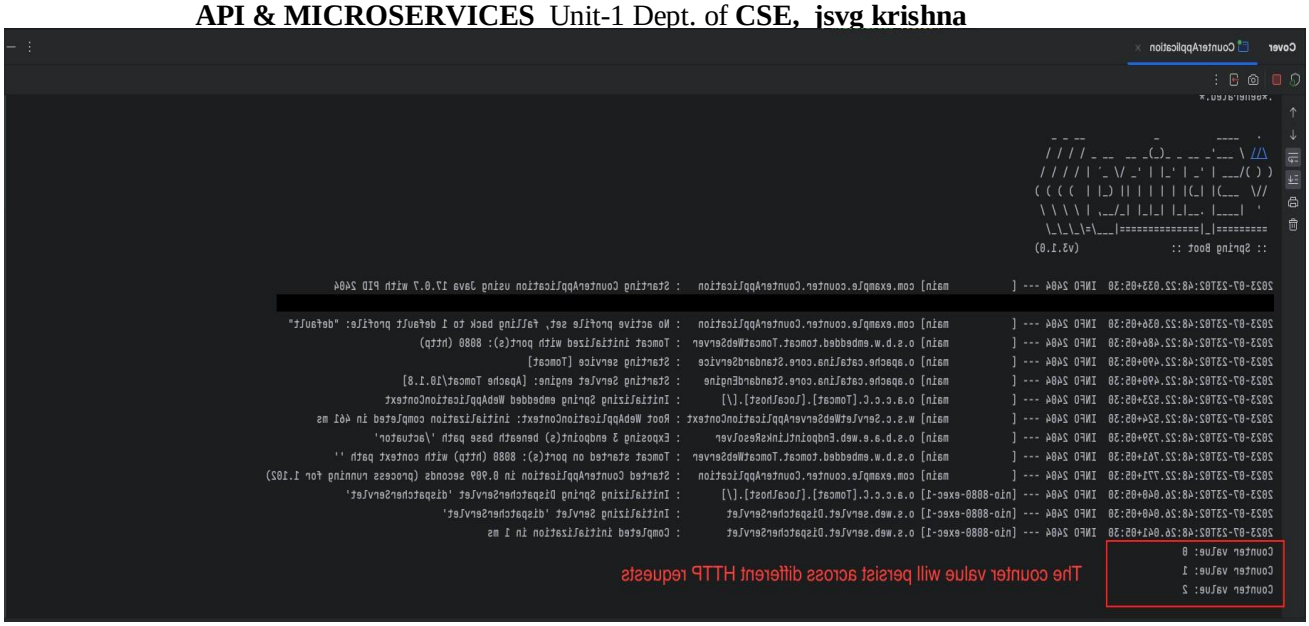
Example:

In this example, we use the custom "application" scope for the `ApplicationBean`. This scope behaves similar to the singleton scope but persists throughout the entire application's lifetime. The `CounterController` uses the `ApplicationBean` instance, and the `getCounter()` method increments the counter each time it's called. Since the "application" scope ensures that there's only one instance of the bean throughout the application, the counter value will persist across different HTTP requests, showing the application scope behavior.

```
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

@Component
@Scope("application")
public class ApplicationBean {
    private int counter = 0;

    public int getCounter() {
        return counter++;
    }
}
```

```
@RestController
public class CounterController {

    @Autowired
    private ApplicationBean applicationBean;

    @GetMapping("/get-counter-value")
    public String counterValue() {
        int currentValue = applicationBean.getCounter();
        System.out.println("Counter value: " + currentValue);

        return "Counter value: " + currentValue;
    }
}
```

Output:



6.WebSocket Scope: Beans for Real-Time WebSockets

- The WebSocket scope is specific to web applications and valid in the context of a web-aware Spring ApplicationContext.
- When a bean is defined with the WebSocket scope, a single instance of the bean is created for each WebSocket connection.
- This scope is useful for managing WebSocket-specific data for individual clients connected to the application.
- WebSocket-scoped beans are ideal for real-time applications using WebSocket communication.

The Working Principle

The WebSocket scope is specific to web applications and creates a single instance of a bean for each WebSocket connection. When a client establishes a WebSocket connection with the server, Spring creates a new instance of the bean associated with the WebSocket scope. The WebSocket-scoped bean instance remains available throughout the entire duration of that specific WebSocket connection. This allows the WebSocket-scoped bean to hold session-specific data and manage WebSocket interactions independently for each client session.

Example:
Let’s consider a chat application where multiple users can connect via WebSocket to participate in real-time chat. Each WebSocket session corresponds to a single user, and we want to maintain user-specific data for each session.

```
import org.springframework.context.annotation.Scope;
import  org.springframework.stereotype.Component;
import org.springframework.web.socket.WebSocketSession;

@Component
@Scope(scopeName = "websocket", proxyMode = ScopedProxyMode.TARGET_CLASS)
public class WebSocketSessionBean {
    private WebSocketSession webSocketSession;
    private String username;

    // Getter and Setter methods for webSocketSession and username

}
```

In this example, we define a `WebSocketSessionBean` with the WebSocket scope. When a new WebSocket connection is established, Spring creates a new instance of this bean and associates it with the WebSocket session. The `WebSocketSessionBean` can then hold the `WebSocketSession` and other user-specific data, such as the username.

With the WebSocket scope, you can manage WebSocket-related state and handle user-specific data during WebSocket interactions, providing a seamless real-time experience for your web application users.

Comparison of Spring Bean Scopes

Let's compare the characteristics of each Spring bean scope.

Scope	Characteristics	Usage
Singleton	– Single shared instance per container	– Global configuration
	– Same instance returned for all requests	– Stateless services
	– Potential thread-safety concerns	– Application-wide beans
Prototype	– New instance per request or injection	– Stateful components
	– Increased memory usage due to multiple instances	– Data-access objects

Scope	Characteristics	Usage
Request	– New instance per HTTP request	– Request-specific data and handlers
	– Isolated for each user request	– User session management
Session	– Single instance per user session	– User-specific data and shopping carts
	– Persists throughout a user's session	– User authentication and session management
Application	– Single instance per web application	– Application-wide settings and constants
	– Shared across all users and components	– Global configuration
WebSocket	– Single instance per WebSocket connection	– Real-time WebSocket communication
	– Specific to web applications	– WebSocket message handling

5. Logger:

What is Logger In Spring Boot?

A logger is an object that allows us to generate log messages within the application. Loggers are part of the logging framework provided by Spring Boot and are used to record various events, activities, and information during the runtime of our application. These log messages provide information into the behavior of our application, assist in troubleshooting, and help in monitoring and maintaining the application's health.

Below are some key points about logger in Spring Boot:

Logger Interface

Spring Boot internally uses the Logger interface from the SLF4J (Simple Logging Facade for Java) library as the primary abstraction for creating log messages. We need to connect with the logger interface to generate log messages in our code.

Logger Initialization

Logger instances are generally initialized at the class level as private static final field. This practice ensures that the logger is shared across instances of the same class and eliminates the overhead of logger creation.

Logger Factory

Spring Boot's logging framework uses a logger factory to create logger instances behind the scenes. The factory is responsible for determining which logging implementation (e.g., Logback, Log4j2) to use and instantiates the appropriate logger accordingly.

Generating Log Messages

In order to generate log messages, we call methods on the logger instance with respect to the desired logging level. For example, to generate an informational log message, you would use logger.info("Message text").

Logging Levels

Logging levels are used to categorize log messages based on their severity and importance. Each logging level corresponds to a specific severity level, which helps developers understand the nature of the logged events. Spring Boot, along with logging frameworks like SLF4J and Logback, provides a set of standard logging levels that are commonly used to indicate different levels of severity. These are **trace**, **debug**, **info**, **warn**, and **error**. Below is the arrangement of the levels based on their severity:

```
trace < debug < info < warn < error
```

By default info & above level messages are enabled.
In order to receive other level messages, go to application.properties and add below entries accordingly:
logger.level.root=TRACE

Since the TRACE has the lowest sevirity, in this case all levels messages will appear.

Log Formatting

Log messages can include placeholders for variables and parameters. The logger framework automatically formats the log messages with the provided values, enhancing readability and making it easier to understand the context of the log entry. However, we can customize the format of the log messages as per our requirement.

Example usage of a logger in Spring Boot application

```
import org.slf4j.Logger;
```

```
import org.slf4j.LoggerFactory;

public class MyService {

  private static final Logger logger =
LoggerFactory.getLogger(MyService.class);

  public void performAction() {

    logger.info("Performing an action...");

    // ... code for performing an action ...

    logger.debug("Action completed successfully.");
  }
}
```

In this example, the LoggerFactory is used to create an instance of the logger for the MyService class. The logger is then used to generate log messages with different logging levels.

6. Introduction to Spring AOP:

AOP is one of the main components in the Spring framework, it provides declarative services for us, such as declarative transaction management (the famous `@Transactional` annotation). Moreover, it offers us the ability to implement custom Aspects and utilize the power of AOP in our applications.

Spring AOP uses either JDK dynamic proxies or CGLIB to create the proxy for a given target object. JDK dynamic proxies are built into the JDK, whereas CGLIB is a common open-source class definition library (repackaged into `spring-core`).

If the target object to be proxied implements at least one interface, a JDK dynamic proxy is used. All of the interfaces implemented by the target type are proxied. If the target object does not implement any interfaces, a CGLIB proxy is created.

AOP Basic Terminologies

The terminologies we will discuss are not Spring specific, they are general AOP concepts that Spring implements.

Let's start by introducing the four main building blocks of any AOP example in Spring.

JoinPoint

Simply put, a JoinPoint is a point in the execution flow of a method where an Aspect (new behavior) can be plugged in.

Advice

It's the behavior that addresses system-wide concerns (logging, security checks, etc…). This behavior is represented by a method to be executed at a JoinPoint. This behavior can be executed Before, After, or Around the JoinPoint according to the Advice type as we will see later.

Pointcut

A Pointcut is an expression that defines at what JoinPoints a given Advice should be applied.

Aspect

Aspect is a class in which we define Pointcuts and Advices.

Spring AOP Example

And now let's put those definitions into a coding example where we create a `Log` annotation that logs out a message to the console before the execution of the method starts.

First, let's include Spring's AOP and test starters dependencies.

```
<dependencies>
        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-test</artifactId>
                <scope>test</scope>
        </dependency>

        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-aop</artifactId>
                <version>2.7.4</version>
        </dependency>
</dependencies>
```

Now, let's create the `Log` annotation we want to use:

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Log {
}
```

What this does is create an annotation that is only applicable to methods and gets processed at runtime.

The next step is creating the Aspect class with a Pointcut and Advice:

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Component;

@Component
@Aspect
public class LoggingAspect {

    @Pointcut("@annotation(Log)")
    public void logPointcut(){
    }

    @Before("logPointcut()")
    public void logAllMethodCallsAdvice(){
        System.out.println("In Aspect");
    }
}
```

Linking this to the definitions we introduced up top we notice the `@Aspect` annotation which marks the `LoggingAspect` class as a source for `@Pointcut` and Advice (`@Before`). Note as well that we annotated the class as a `@Component` to allow Spring to manage this class as a Bean.

Moreover, we used the expression `@Pointcut("@annotation(Log)")` to describe which potential methods (JoinPoints) are affected by the corresponding Advice method. In this case, we want to add the advice to all methods that are annotated with our `@Log` annotation.

This brings us to `@Before("logPointcut()")` that executes the annotated method `logAllMethodCallsAdvice` before the execution of any method annotated with `@Log`.

Now, let’s create a Spring Service that will use the aspect we defined:

```
import org.springframework.stereotype.Service;

@Service
public class ShipmentService {
  @Log
  // this here is what's called a join point
  public void shipStuff(){
       System.out.println("In Service");
  }
}
```

And let's test it out in a `@SpringBootTest`

```
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest
class AopApplicationTests {
    @Autowired
    ShipmentService shipmentService;

    @Test
    void testBeforeLog() {
       shipmentService.shipStuff();
    }
}
```

This will spin up a Spring context and load the `LoggingAspect` and the `ShipmentService`. Next, in the test method, we call the `shipStuff()` method which was annotated by `@Log`.

If we check the console we should see

```
In Aspect
In Service
```

This means that the `logAllMethodCallsAdvice` method was indeed executed before the `shipStuff()` method.

7. Implementing AOP advices:

@Before

We can capture the JoinPoint at the `@Before` annotated method which offers us much useful information like the method name, method arguments, and many more. For example, let's log the name of the method.

```
@Component
@Aspect
public class LoggingAspect {
```

```
    @Pointcut("@annotation(Log)")
    public void logPointcut(){}

    @Before("logPointcut()")
    public void logAllMethodCallsAdvice(JoinPoint joinPoint){
        System.out.println("In Aspect at " + joinPoint.getSignature().getName());
    }
}
```

And testing it

```
@SpringBootTest
class AopApplicationTests {
    @Autowired
    ShipmentService shipmentService;

    @Test
    void testBeforeLog() {
        shipmentService.shipStuff();
    }
}
```

Will print out

```
In Aspect at shipStuff
In Service
```

@After

This advice is run after the method finishes running, this could be by normally returning or by throwing an exception.

Let’s introduce a new annotation

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface AfterLog {}
@Component
@Aspect
public class LoggingAspect {
            ...
    @Pointcut("@annotation(AfterLog)")
    public void logAfterPointcut(){}

    @After("logAfterPointcut()")
    public void logMethodCallsAfterAdvice(JoinPoint joinPoint) {
        System.out.println("In After Aspect at " + joinPoint.getSignature().getName());
    }
}
```

And let’s modify our service to use the new annotation

```
@Service
public class OrderService {
            ...
```

```
    @AfterLog
    public void checkStuff() {
        System.out.println("Checking stuff");
    }
}
```

And as for the test

```
@SpringBootTest
class AopApplicationTests {
                ...
    @Test
    void testCheckingStuffWithAfter() {
        orderService.checkStuff();
    }
}
```

This should output

```
Checking stuff
In After Aspect at checkStuff
```

@AfterReturning

This is similar to `@After` but it's run only after a normal execution of the method.

@AfterThrowing

This is similar to `@After` but it's run only after an exception is thrown while executing the method.

@Around

This annotation allows us to take actions either before or after a JoinPoint method is run. We can use it to return a custom value or throw an exception or simply let the method run and return normally.

Let's start by defining a new `ValidationService`

```
@Service
public class ValidationService {
    public void validateNumber(int argument) {
        System.out.println(argument + " is valid");
    }
}
```

And a new Aspect class

```
@Component
@Aspect
```

```
public class ValidationAspect {
    @Pointcut("within(io.reflectoring.springboot.aop.ValidationService)")
    public void validationPointcut(){}

    @Around("validationPointcut()")
    public void aroundAdvice(ProceedingJoinPoint joinPoint) throws Throwable {
        System.out.println("In Around Aspect");
        int arg = (int) joinPoint.getArgs()[0];
        if (arg < 0)
            throw new RuntimeException("Argument should not be negative");
        else
            joinPoint.proceed();
    }
}
```

The above Pointcut expression will capture all methods that are in the class ValidationService. Then, the aroundAdvice() advice will check the first argument of the method if it's negative it will throw an exception, otherwise it will allow the method to execute and return normally.

```
@SpringBootTest
class AopApplicationTests {
                ...
    @Autowired
    ValidationService validationService;

    @Test
    void testValidAroundAspect() {
        validationService.validateNumber(10);
    }
}
```

This will print out

```
In Around Aspect
10 is valid
```

And now let's try a case where we will get an exception.

```
@SpringBootTest
class AopApplicationTests {
                ...
    @Autowired
    ValidationService validationService;
                ...
    @Test
    void testInvalidAroundAspect() {
        validationService.validateNumber(-4);
    }
}
```

This should output

```
In Around Aspect

java.lang.RuntimeException: Argument should not be negative
...
```

8. Best Practices : Spring Boot Application:

1. Standard Project Structure for Spring Boot Projects

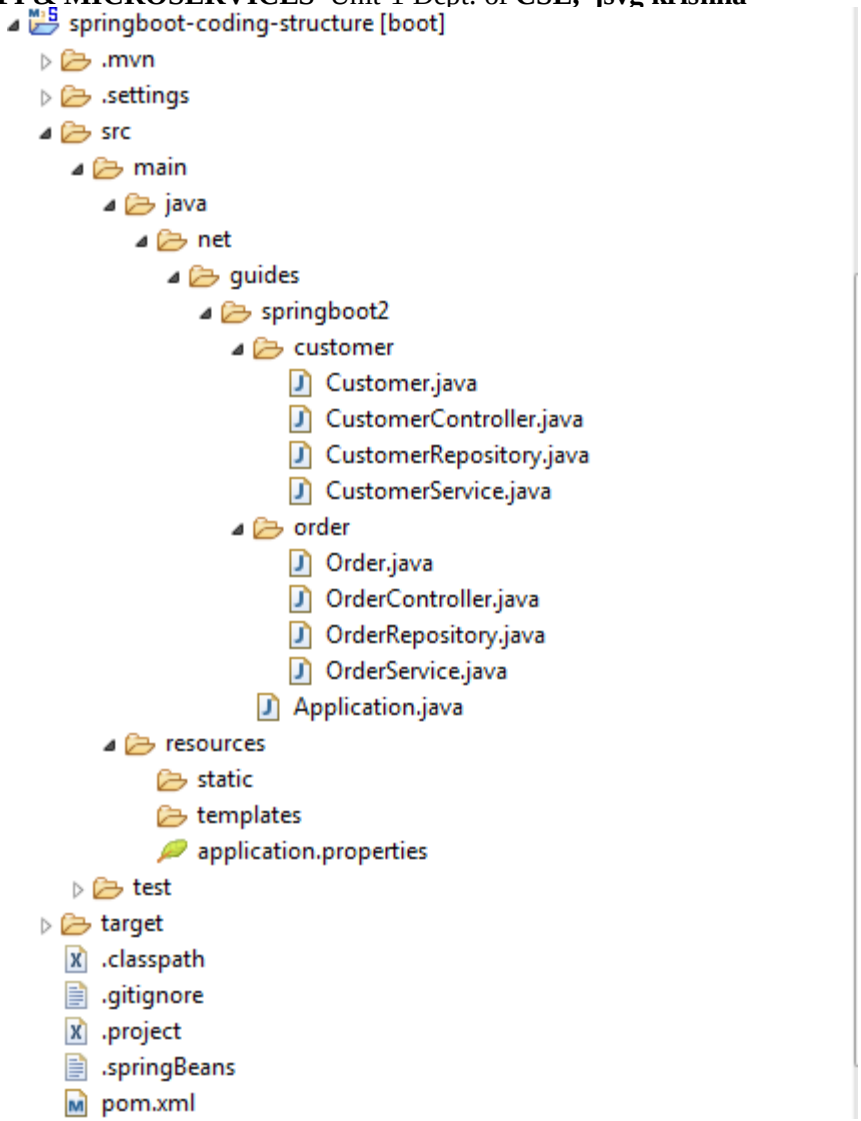
Here, I will show you recommended ways to create a spring boot project structure in Spring boot applications, which will help beginners to maintain standard coding structure.

Don't use the "default" Package

When a class does not include a package declaration, it is considered to be in the "default package". The use of the "default package" is generally discouraged and should be avoided. It can cause particular problems for Spring Boot applications that use the `@ComponentScan`, `@EntityScan`, or **@SpringBootApplication** annotations since every class from every jar is read.
I recommend that you follow Java's recommended package naming conventions and use a reversed domain name (for example, com.javaguides.projectname).
To know standard package naming conventions, check out - about **Java Packages with Examples**.
Here, I recommend two approaches to create a standard project structure in Spring boot applications.

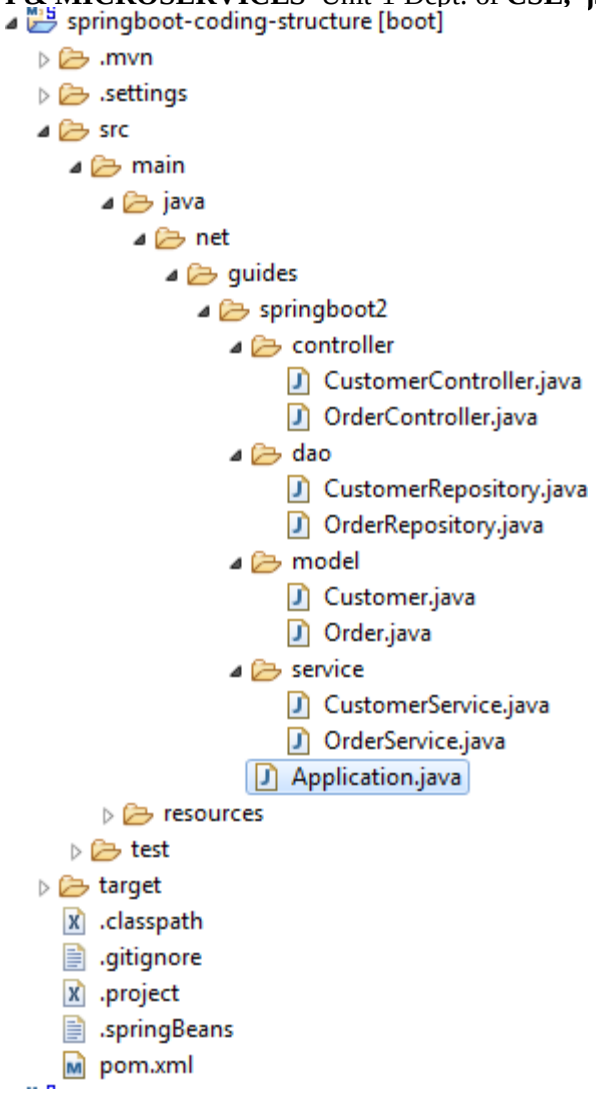
Project Structure - First approach

The following spring boot project structure shows a typical layout recommended by spring boot team:



Project Structure - Second approach

However, the above typical layout approach works well but some of the developers prefer to use the following packaging or project structure:



Separate package for each layer like a model, controller, dao and service etc.

2. Using Java-based configuration - @Configuration

Spring Boot favors Java-based configuration. Although it is possible to use *SpringApplication* with XML sources, I generally recommend that your primary source be a single **@Configuration** class. Usually, the class that defines the main method is a good candidate as the primary **@Configuration** or @SpringBootApplication annotation.

```
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
         SpringApplication.run(Application.class, args);
    }
}
```

It is always a good practice to divide a large Spring-based configuration file into multiple files as per your convenience. I generally keep Database configuration in one file (DatabaseConfig), Spring

MVC beans in another file, etc, and then finally import in the main file.

The below example demonstrates the same here.

Importing Additional Configuration Classes

You need not put all your *@Configuration* into a single class. The @Import annotation can be used to import additional configuration classes.
For example:

```
@Configuration
public class DataSourceConfig {
    @Bean
    public DataSource dataSource() {
        return new DriverManagerDataSource(...);
    }
}
```

Let's import above additional configuration into *AppConfig* class:

```
@Configuration
@AnnotationDrivenConfig
@Import(DataSourceConfig.class) // <-- AppConfig imports DataSourceConfig
public class AppConfig extends ConfigurationSupport {
    @Autowired DataSourceConfig dataSourceConfig;

    @Bean
    public void TransferService transferService() {
        return new TransferServiceImpl(dataSourceConfig.dataSource());
    }
}
```

Multiple configurations may be imported by supplying an array of classes to the *@Import* annotation:

```
@Configuration
@Import({ DataSourceConfig.class, TransactionConfig.class })
public class AppConfig extends ConfigurationSupport {
    // @Bean methods here can reference @Bean methods in DataSourceConfig or
TransactionConfig
}
```

3. Using Auto-configuration

Spring Boot auto-configuration attempts to automatically configure your Spring application based on the jar dependencies that you have added.
The simplest way to make use of it is to rely on the Spring Boot Starters. So, if you want to interact

with Redis, you can start by including:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

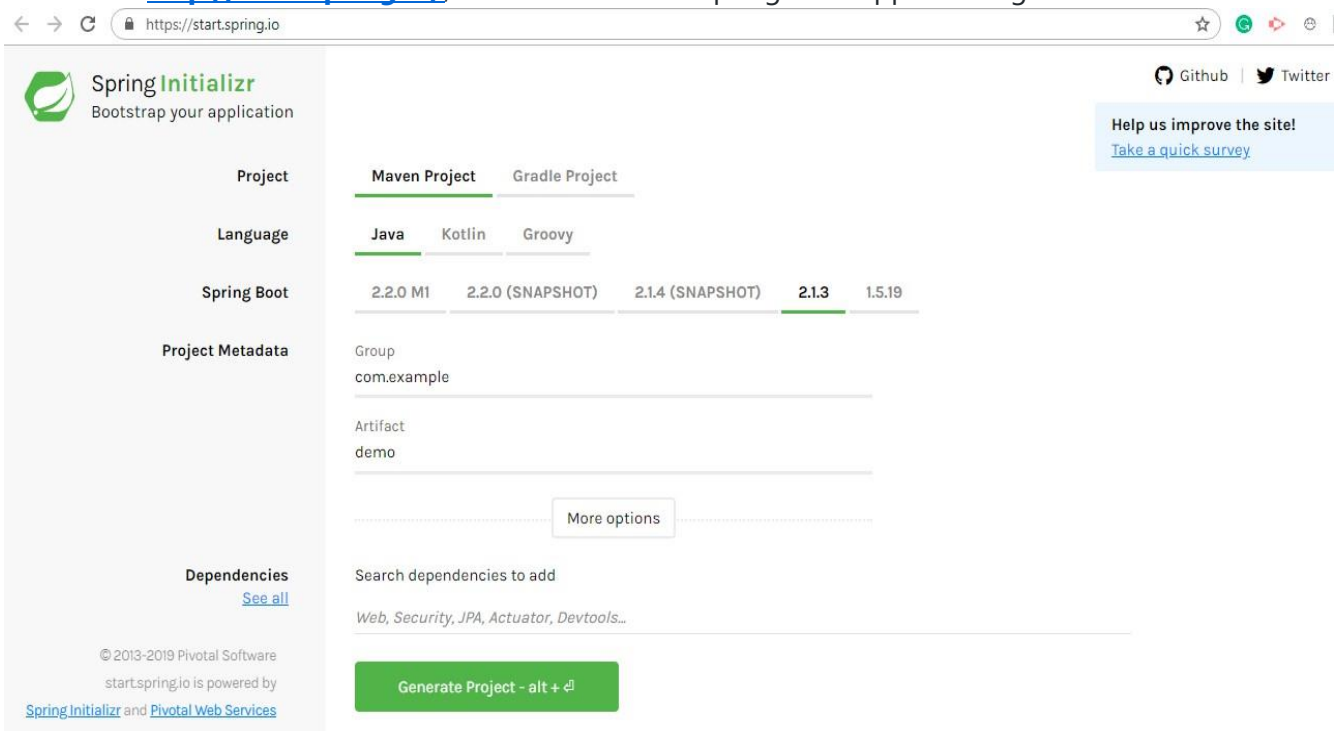
if you want to work with MongoDB, you have:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
```

and so on… By relying on these starters, you are relying on a tested and proven configuration that is going to work well together.
It is possible to exclude some classes from the Auto-configuration, by using the following annotation
property: *@EnableAutoConfiguration(exclude={ClassNotToAutoconfigure.class})*, but you should do it only if absolutely necessary.

4. Use Spring Initializr for starting new Spring Boot projects

There are many ways to create a **Spring Boot application**. The simplest way is to use Spring Initializr at **http://start.spring.io/**, which is an online Spring Boot application generator.



5. When to Use Constructor-based and setter-based DI in Spring or Spring boot applications

As we know that there are three possible ways of DI in Spring:

1. Constructor-based dependency injection
2. Setter-based dependency injection
3. Field-based dependency injection

From **spring Framework documentation**, since you can mix **constructor-based** and **setter-based** DI, it is a good rule of thumb to use constructors for mandatory dependencies and setter methods or configuration methods for optional dependencies.

As I can see many developers prefer to use constructor injection over setter based injection because this makes bean class object as immutable(We cannot change the value by Constructor injection).

For example, define dependencies as final variables like:

```
@Service
@Transactional
public class ProductService {
    private final ProductRepository productRepository;

    public ProductService(ProductRepository productRepository) {
        this.productRepository= productRepository;
    }
}
```

This way it will be easier to instantiate *ProductService* with necessary dependencies for testing if required.

6. Using @Service Annotation Class for Business Layer

I observed in many of the spring boot open-source applications, the Spring Data JPA repositories directly called in Controller classes in Spring boot applications. I generally use Service class annotated with a @Service annotation to write business-related logic so you can follow this approach.
For example, UserService, CustomerService, EmployeeService etc.

I listed a few free spring boot projects here at **10+ Free Open Source Projects Using Spring Boot** article.

7. Spring Bean Naming Conventions

The convention is to use the standard Java convention for instance field names when naming beans. That is, bean names start with a lowercase letter, and are camel-cased from then on. Examples of such names would be (without quotes) '**accountManager**', '**accountService**', '**userDao**', '**loginController**', and so forth.

Naming beans consistently make your configuration easier to read and understand, and if you are using Spring AOP it helps a lot when applying advice to a set of beans related by name.

Spring Bean Naming Conventions in Java-based config

```
@Configuration
public class Application {

 @Bean(name = "cService")
  public CustomerService customerService() {
   return new CustomerService();
  }

 @Bean(name = "oService")
  public OrderService orderService() {
   return new OrderService();
  }
 }
```

Spring Bean Naming Conventions in an annotation-based config

```
@Service("emailService")
public class EmailService implements MessageService{
    public void sendMsg(String message) {
        System.out.println(message);
    }
}
```

```
@Service("twitterService")
public class TwitterService implements MessageService{
    public void sendMsg(String message) {
        System.out.println(message);
    }
}
```

Reference: **https://docs.spring.io/spring/docs/current/spring-framework-**

reference/core.html#beans-definition

8. Handle Circular dependencies

If you use predominantly constructor injection, it is possible to create an unresolvable circular dependency scenario.
For example - Class **A** requires an instance of class **B** through constructor injection, and class **B** requires an instance of class **A** through constructor injection. If you configure beans for classes **A** and **B** to be injected into each other, the Spring IoC container detects this circular reference at runtime and throws a *BeanCurrentlyInCreationException*.

```
public  class A {
    public A(B b){
        ....
    }
}
public  class B {
    public B(A b){
        ....
    }
}
```

One possible solution is to edit the source code of some classes to be configured by setters rather than constructors. Alternatively, avoid constructor injection and use setter injection only. In other words, although it is not recommended, you can configure circular dependencies with setter injection.

9. Exception Handling in Spring boot Rest API projects

Handle proper exceptions and custom error messages for RESTful Web Services developed using Spring Boot.

Check out this article - **Spring Boot 2 Exception Handling for REST APIs**.

10. Follow Restful API Design Best Practices in Spring Boot Rest API Applications

I have written a separate **article of Restful API design best practices**. Here is a list of best practices to design a clean RESTful API.

API & MICROSERVICES

IMPORTANT QUESTIONS - UNIT-2

1. Explain about Spring Boot: Creating a Spring Boot Application?
2. Explain Spring Boot Application Annotation?
3. What is Autowiring?
4. Scope of a bean?
5. Explain about Logger
6. Introduction to Spring AOP
7. Write aShort note on Implementing AOP advices?
8. Explain Best Practices : Spring Boot Application