**Name-Hritik Kumar Sharma**

**QID-24510066  
Course- MCA-3rd Sem (2025-2026)**

**1.Difference between Spring and Spring Boot.**

**Spring**

[**Spring**](https://www.geeksforgeeks.org/advance-java/introduction-to-spring-framework/) is an open-source lightweight framework that allows Java developers to build simple, reliable, and scalable enterprise applications. This framework mainly focuses on providing various ways to help you manage your business objects. It made the development of Web applications much easier compared to classic Java frameworks and Application Programming Interfaces (APIs), such as Java database connectivity (JDBC), JavaServer Pages (JSP), and Java Servlet. This framework uses various new techniques such as Aspect-Oriented Programming (AOP), Plain Old Java Object (POJO), and dependency injection (DI), to develop enterprise applications. The **Spring framework** can be considered as a collection of sub-frameworks, also called layers, such as Spring AOP. Spring Object-Relational Mapping (Spring ORM). Spring Web Flow, and Spring Web MVC. You can use any of these modules separately while constructing a Web application. The modules may also be grouped to provide better functionalities in a Web application.

**Spring Boot**

[**Spring Boot**](https://www.geeksforgeeks.org/springboot/introduction-to-spring-boot/) is built on top of the conventional spring framework. So, it provides all the features of spring and is easier to use than spring. Spring Boot is a microservice-based framework and makes a production-ready application in very less time. In Spring Boot everything is auto configured. We just need to use proper configuration for utilizing a particular functionality. Spring Boot is very useful if we want to develop REST API.

**Why Spring Boot over Spring?**

Let us know, if Spring was solving all the problems, then what is the need for Spring Boot at all?

The reason why we need Spring Boot is we are changing or shifting towards applications like microservices and with microservices, one of the most important thing aim is we would want to be able to develop applications very quickly. So instead of building one large application, we would like to build ten small microservices, which have their own scope and their own capabilities. Spring-based applications have lots of configurations. It can be of **XML configuration**, Java configuration or annotations, etc. For example, If we want to use Spring MVC, we need to use **@ComponentScan** annotation, **Dispatcher servlet**, **view resolver**, **web jars,** etc. This kind of configuration makes it slow to develop an application. So, in this place, **Spring Boot Autoconfiguration** comes in. It looks at what types of frameworks are available at the class path and it looks at what configurations are provided by the programmers or what configurations are provided already for the application. It will look at both of them. Data is not configured but there is hibernation on the class path, so it will configure the data source automatically. It will configure the **in-memory**database, it will configure the dispatcher servlet automatically. This is called autoconfiguration. Spring Boot creates a starter project by which all the XML configurations and dependencies get by default.

**Difference between Spring and Spring Boot**

| **Spring** | **Spring Boot** |
| --- | --- |
| Spring is an open-source lightweight framework widely used to develop enterprise applications. | Spring Boot is built on top of the conventional spring framework, widely used to develop REST APIs. |
| The most important feature of the Spring Framework is dependency injection. | The most important feature of the Spring Boot is Autoconfiguration. |
| It helps to create a loosely coupled application. | It helps to create a stand-alone application. |
| To run the Spring application, we need to set the server explicitly. | Spring Boot provides embedded servers such as Tomcat and Jetty etc. |
| To run the Spring application, a deployment descriptor is required. | There is no requirement for a deployment descriptor. |
| To create a Spring application, the developers write lots of code. | It reduces the lines of code. |
| It doesn't provide support for the in-memory database. | It provides support for the in-memory database such as H2. |
| Developers need to write boilerplate code for smaller tasks. | In Spring Boot, there is reduction in boilerplate code. |
| Developers have to define dependencies manually in the pom.xml file. | pom.xml file internally handles the required dependencies. |

**2.How Spring Boot Application Works Internally?**

Spring Boot Application starts using a main method, like any other Java program, and the main method is called the "**run**" method i.e. *SpringApplication.run()*,. From this run method, the application context of [IOC (Inversion Of Control)](https://www.geeksforgeeks.org/advance-java/spring-understanding-inversion-of-control-with-example/)searches the class annotated with @Configuration annotation which calls all the beans in the classpath and initializes those classes. Beans are stored inside a particular space in[JVM (Java Virtual Machine)](https://www.geeksforgeeks.org/java/how-jvm-works-jvm-architecture/). That particular space is known as the **IOC Container**.

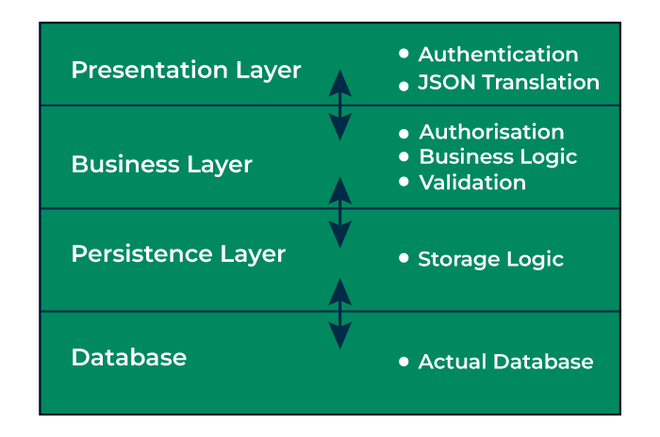
After beans are created the requests will go to the [dispatcher servlet](https://www.geeksforgeeks.org/java/what-is-dispatcher-servlet-in-spring/)and the dispatcher servlet will distribute all the requests among the appropriate controllers. To understand how a Spring Boot application works internally, it is essential to know its key components and the flow of the application.

**Spring Boot Layered Architecture**

Spring Boot follows a layered architecture in which each layer communicates with the other layer directly in a hierarchical structure.

There are 4 layers in Spring Boot as follows:

1. Presentation Layer
2. Business Layer
3. Persistence Layer
4. Database Layer

****

**1. Presentation Layer**

This layer handles all the HTTP requests made by clients, then it translates the JSON parameter to object and also authenticates the request, and transfer it to the business layer. In short, it only consists of the frontend part or we can say the view part.

**2. Business Layer**

This is also known as the Service layer which handles all the business logic of an application. It consists of service classes and uses services provided by the data access layers. It also performs authorization and validation.

**3. Persistence Layer**

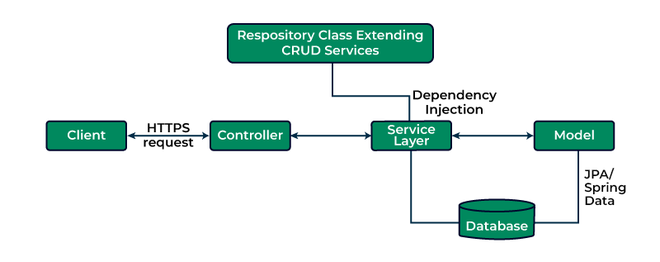
It contains all the storage logic which are required and translates the business objects to database rows.

**4. Database Layer**

In this layer, all the CRUD (create, read update, delete) operations are performed.

**Spring Boot Application Flow Architecture**

Let us understand the control flow of a Spring Boot Application in the below diagram:



**Explanation:**

1. Client makes an HTTP request(GET, POST, PUT, DELETE) to the browser.
2. Then the request will go to the controller where all the requests will be mapped and handled.
3. After mapping done, in Service layer all the business logic will be performed. It performs the logic on the data that is mapped to JPA(Java Persistence API) using model classes.
4. In repository layer, all the CRUD operations is done for the rest APIs.
5. A JSP page is returned to the user if no errors are there.

**How Does Spring Boot Application Starts?**

Spring Boot is very popular for It's auto-configuration feature. This means it automatically configures our application based on the dependencies added during the project creation. As we don't need to configure xml file, Spring internally do all the xml configuration which are provided by jar. We just have to use the pre-configured jars. To enable those pre-configured jars, simply we need to define starter dependencies in pom.xml file.

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

By adding this **starter-data-jpa dependency**, we get all the pre-configured JPA configuration which are needed for connecting our application with the database. Then we just need to focus on defining the JPA entities, repositories, business logic, while Spring Boot takes care of all internal configuration and database interaction.

**Basic Annotations to Start a Spring Boot Application**

The entry point of the Spring Boot Application is the class which contains **@SpringBootApplication** annotation along with the main method. The main method should contain *SpringApplication.run* method.

**Below is the implementation of the above method:**

**import** **org.springframework.boot.SpringApplication**;

@SpringBootApplication

**public** **class** **GFG** {

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

SpringApplication.run(GFG.class, args);

}

}

Here we have used @SpringBootApplication annotation along with the class GFG and a main method just like other java program with *SpringApplication.run* method to bootstrap the application.

**How @SpringBootApplication Works?**

@SpringBootApplication annotation is the combination of another three annotations i.e.[@Configuration](https://www.geeksforgeeks.org/springboot/spring-configuration-annotation-with-example/), @EnableAutoConfiguration, [@ComponentScan](https://www.geeksforgeeks.org/springboot/spring-componentscan-annotation-with-example/).

*@SpringBootApplication = @Configuration + @EnableAutoConfiguration + @ComponentScan*

**@Configuration Annotation:**

* This annotation configures the application context such as transaction, resource handler, view resolver, security etc.
* It is used in class level and it specifically indicates that a class declare one or more than one @Bean methods.

**Below is the implementation of the above method:**

@Configuration

**public** **class** **GFG** {

@Bean(name = gfg)

**public** demoClass demo(){

}

}

**@EnableAutoConfiguration Annotation:**

* This annotation will automatically configures our application we don't need to configure manually.
* It enables the auto-configuration feature of Spring Boot.

**Below is the implementation of the above method:**

**import** **org.springframework.boot.SpringApplication**;

**import** **org.springframework.boot.autoconfigure.EnableAutoConfiguration**;

*// Annotaion used*

@EnableAutoConfiguration

**public** **class** **GFG** {

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

}

}

Here we have used @EnableAutoConfiguration annotation along with the class name to perform the automatic configuration over an application.

**@ComponentScan Annotation:**

* This annotation will automatically scans all the beans and package declaration when the application initializes inside the class path.
* It will automatically scan all the components added to our project.

**Below is the implementation of the above method:**

*// @ComponentScan Annotation*

@ComponentScan(&quot;com.geeksforgeeks.springboot&quot;)

@SpringBootApplication

**public** **class** **GFG** {

}

Here we have used @ComponentScan annotation along with the @SpringBootApplication annotation which will scan the package which is passed inside the parameter. We can also use this annotation without argument, where Spring is responsible for scan the current package and it's sub-packages. We can use these annotations separately as per our need because spring boot supports loosely coupled feature.

Spring Boot enables the developers to use a single annotation instead of using three different annotations we can use one annotation which has the default attributes of all those annotations.

**3. Spring Boot - Auto-configuration.**

Spring Boot is heavily attracting developers toward it because of three main features as follows:

1. Auto-configuration - such as checking for the dependencies, the presence of certain classes in the classpath, the existence of a bean, or the activation of some property.
2. An opinionated approach to configuration.
3. The ability to create stand-alone applications.

**Auto-Configuration in Spring Boot**

* @Conditional annotation acts as a base for the Spring Boot auto-configuration annotation extensions.
* It automatically registers the beans with @Component, @Configuration, @Bean, and meta-annotations for building custom stereotype annotations, etc.
* The annotation @EnableAutoConfiguration is used to enable the auto-configuration feature.
* The @EnableAutoConfiguration annotation enables the auto-configuration of Spring ApplicationContext by scanning the classpath components and registering the beans.
* This annotation is wrapped inside the @SpringBootApplication annotation along with @ComponentScan and @SpringBootConfiguration annotations.
* When running main() method, this annotation initiates auto-configuration.

**Implementation:**Bootstrapping of Application

**package** **gfg**;

**import** **org.springframework.boot.SpringApplication**;

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

@SpringBootApplication

**public** **class** **GfgApplication** {

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

{

SpringApplication.run(GfgApplication.class, args);

}

}

***Note****: You should use the '@EnableAutoConfiguration' annotation only one time in your application.*

* *'spring-boot-autoconfigure.jar' is the file that looks after all the auto-configuration.*
* *All auto-configuration logic for MVC, data, JMS, and other frameworks is present in a single jar*

A diagram of a application

AI-generated content may be incorrect.

**Working of Auto-Configuration in Spring Boot**

**A: Dependencies**

* Auto-Configuration is the main focus of the Spring Boot development.
* Our Spring application needs a respective set of dependencies to work.
* Spring Boot auto-configures a pre-set of the required dependencies without a need to configure them manually.
* This greatly helps and can be seen when we want to create a stand-alone application.
* When we build our application, Spring Boot looks after our dependencies and configures both the underlying Spring Framework and required jar dependencies (third-party libraries ) on the classpath according to our project built.
* It helps us to avoid errors like mismatches or incompatible versions of different libraries.
* If you want to override these defaults, you can override them after initialization.

**Tool: Maven**

**Example 1: pom.xml**

<?xml version="1.0" encoding="UTF-8"?>

**<project** xmlns="https://maven.apache.org/POM/4.0.0"

xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd"**>**

**<modelVersion>**4.0.0**</modelVersion>**

**<parent>**

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

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

**<version>**2.5.6**</version>**

**<relativePath/>** *<!-- lookup parent from repository -->*

**</parent>**

**<groupId>**sia**</groupId>**

**<artifactId>**taco-cloud**</artifactId>**

**<version>**0.0.1-SNAPSHOT**</version>**

**<name>**taco-cloud**</name>**

**<description>**Demo project for Spring Boot**</description>**

**<properties>**

**<java.version>**11**</java.version>**

**<vaadin.version>**14.7.5**</vaadin.version>**

**</properties>**

**<dependencies>**

**<dependency>**

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

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

**</dependency>**

**<dependency>**

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

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

**</dependency>**

**<dependency>**

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

**<artifactId>**spring-boot-devtools**</artifactId>**

**<scope>**runtime**</scope>**

**<optional>**true**</optional>**

**</dependency>**

**<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-jersey**</artifactId>**

**</dependency>**

**<dependency>**

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

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

**</dependency>**

**<dependency>**

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

**<artifactId>**spring-boot-starter-webflux**</artifactId>**

**</dependency>**

**<dependency>**

**<groupId>**com.vaadin**</groupId>**

**<artifactId>**vaadin-spring-boot-starter**</artifactId>**

**</dependency>**

**<dependency>**

**<groupId>**io.projectreactor**</groupId>**

**<artifactId>**reactor-test**</artifactId>**

**<scope>**test**</scope>**

**</dependency>**

**<dependency>**

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

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

**</dependency>**

**<dependency>**

**<groupId>**org.projectlombok**</groupId>**

**<artifactId>**lombok**</artifactId>**

**<optional>**true**</optional>**

**</dependency>**

**<dependency>**

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

**<artifactId>**spring-boot-starter-data-jdbc**</artifactId>**

**</dependency>**

**<dependency>**

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

**<artifactId>**spring-boot-starter-jdbc**</artifactId>**

**</dependency>**

**<dependency>**

**<groupId>**com.h2database**</groupId>**

**<artifactId>**h2**</artifactId>**

**<scope>**runtime**</scope>**

**</dependency>**

**<dependency>**

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

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

**</dependency>**

**<dependency>**

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

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

**</dependency>**

**<dependency>**

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

**<artifactId>**spring-security-test**</artifactId>**

**<scope>**test**</scope>**

**</dependency>**

**<dependency>**

**<groupId>**mysql**</groupId>**

**<artifactId>**mysql-connector-java**</artifactId>**

**<scope>**runtime**</scope>**

**</dependency>**

**<dependency>**

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

**<artifactId>**spring-boot-starter-hateoas**</artifactId>**

**</dependency>**

**<dependency>**

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

**<artifactId>**spring-boot-starter-data-rest**</artifactId>**

**</dependency>**

**</dependencies>**

**<build>**

**<plugins>**

**<plugin>**

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

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

**<configuration>**

**<excludes>**

**<exclude>**

**<groupId>**org.projectlombok**</groupId>**

**<artifactId>**lombok**</artifactId>**

**</exclude>**

**</excludes>**

**</configuration>**

**</plugin>**

**</plugins>**

**</build>**

**<dependencyManagement>**

**<dependencies>**

**<dependency>**

**<groupId>**com.vaadin**</groupId>**

**<artifactId>**vaadin-bom**</artifactId>**

**<version>**${vaadin.version}**</version>**

**<type>**pom**</type>**

**<scope>**import**</scope>**

**</dependency>**

**</dependencies>**

**</dependencyManagement>**

**<profiles>**

**<profile>**

**<id>**production**</id>**

**<build>**

**<plugins>**

**<plugin>**

**<groupId>**com.vaadin**</groupId>**

**<artifactId>**vaadin-maven-plugin**</artifactId>**

**<version>**${vaadin.version}**</version>**

**<executions>**

**<execution>**

**<id>**frontend**</id>**

**<phase>**compile**</phase>**

**<goals>**

**<goal>**prepare-frontend**</goal>**

**<goal>**build-frontend**</goal>**

**</goals>**

**<configuration>**

**<productionMode>**true**</productionMode>**

**</configuration>**

**</execution>**

**</executions>**

**</plugin>**

**</plugins>**

**</build>**

**</profile>**

**</profiles>**

**</project>**

**Understanding Auto-Configuration of Dependencies**

* When you build a Spring Boot project, the 'Starter Parent' dependency gets automatically added in the 'pom.xml' file.
* It notifies that the essential 'sensible' defaults for the application have been auto-configured and you therefore can take advantage of it.

<parent>

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

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

<version>...</version>

</parent>

* To add the dependency ( library of tech stacks ), you don't need to mention the version of it because the Spring Boot automatically configures it for you.
* Also, when you update/change the Spring Boot version, all the versions of added dependencies will also get updated/changed.

<dependency>

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

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

</dependency>

* It is Spring Boot's auto-configuration that makes managing dependencies supremely easy for us.
* With the help of enabling 'debug logging' in the 'application.properties' file, we can know more about auto-configuration.

logging.level.org.springframework: DEBUG

**Tool B: Gradle**

**Example 2: build.gradle**

buildscript {

repositories {

jcenter()

}

dependencies {

classpath("org.springframework.boot:spring-boot-gradle-plugin:1.3.8.RELEASE")

}

}

apply plugin: 'java'

apply plugin: 'spring-boot'

repositories {

jcenter()

}

dependencies {

compile("org.springframework.boot:spring-boot-starter-web")

testCompile("org.springframework.boot:spring-boot-starter-test")

}

**B:** Spring Application

**Illustration:** Class

1. @Bean is a method-level annotation.
2. @Bean annotation specifies that a method produces a return value registered as a bean ( data ) with BeanFactory - managed by Spring Container.
3. This particular java program uses @Configuration annotation specifying that the class contains one or more @Bean annotations which help to automatically register (initialize) in the Spring Container (Spring Application Context).
4. @Configuration is a class-level annotation.

**Example**

*// DataSourceConfiguration of Data Source*

**package** **gfg**;

**import** **javax.sql.DataSource**;

**import** **org.springframework.boot.jdbc.DataSourceBuilder**;

**import** **org.springframework.context.annotation.Bean**;

**import** **org.springframework.context.annotation.Configuration**;

@Configuration

**public** **class** **ConfigDataSource** {

@Bean **public** **static** DataSource source()

{

DataSourceBuilder<?> dSB

= DataSourceBuilder.create();

dSB.driverClassName("com.mysql.jdbc.Driver");

dSB.url("jdbc:mysql://localhost:3306/userdetails");

dSB.username("user");

dSB.password("password");

**return** dSB.build();

}

}

**4.Properties with Spring and Spring Boot.**

Java-based applications using the Spring framework and its evolution into the Spring Boot and the properties play a crucial role in configuring the various aspects of the application. Properties can allow the developers to externalize the configuration settings from the code. Understanding how to work with properties in the Spring and Spring Boot is essential for building flexible and maintainable Spring applications.

**Prerequisites**

* Good Understanding of Java programming
* Basic understanding of Spring framework and Spring Boot basics
* JDK and Spring environment setup into your local system

**Understanding of the Properties of Spring and Spring Boot**

Properties in the Spring and Spring Boot are typically stored in the configuration files such as the `**application.properties`** or `**application.yml`**. These files reside in the src/main/resources directory of the Spring Boot application.

Properties can be used to configure the various aspects of the application such as the database connection details, logging settings, server port, etc. It can be injected into the Spring Components using the @Value annotation or access programmatically using the Environment abstraction of the Spring application.

**Key Terminologies**

* **Properties**: It can be key-value used for the configuring the various aspects of the application. These properties can typically stored in the configuration files of the Spring application.
* **Configuration Files**: It can be files where the properties are stored. In the Spring Boot, Commonly used for the configuration files are **'application.properties'** or '**application.yml'**. These files are reside in the src/main/resources directory.
* **@Value annotation**: This annotation can be used for the injecting property values into the Spring beans. It can be used to the directly inject the values from the properties files into the beans fields or the constructors parameters.

**Project to Implement the Properties with Spring and Spring Boot**

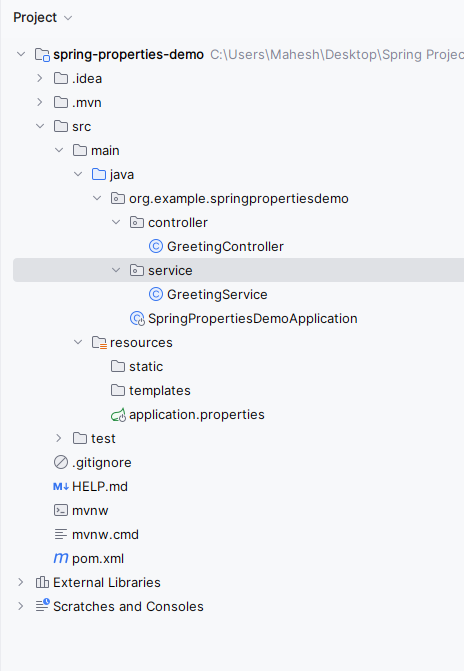
We can demonstrate how to the Implement the Properties with Spring and Spring Boot in the Spring application.

**Step 1**: Create the spring project using Spring STS IDE including the below mentioned dependencies into the project.

**Dependencies:**

* Spring Web
* Lombok
* Spring dev tools

Once the the project creation completed successfully, it will look like below image,



**Step 2**: Open the **application.properties** file and put the below code for the server port and spring security credentials configuration to the project.

spring.application.name=spring-properties-demo  
app.greeting.message=Hello, Spring Boot!

**Step 3**: Create the new package and it named as the **service**in that package create the new Java class and it named as **GreetingService**. Go to **src > org.example.springpropertiesdemo > service > GreetingService** and put the below code.

package org.example.springpropertiesdemo.service;

​

import org.springframework.beans.factory.annotation.Value;

import org.springframework.stereotype.Service;

​

@Service

public class GreetingService {

​

@Value("${app.greeting.message}")

private String greetingMessage;

​

public String getGreetingMessage() {

return greetingMessage;

}

}

**Step 4**: Create the new package and it named as the **controller**in that package create the new Java class and it named as **GreetingController**. Go to **src > org.example.springpropertiesdemo > service > GreetingController**and put the below code.

package org.example.springpropertiesdemo.controller;

​

import org.example.springpropertiesdemo.service.GreetingService;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.stereotype.Service;

​

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RestController;

​

@RestController

public class GreetingController {

​

@Autowired

private GreetingService greetingService;

​

@GetMapping("/greeting")

public String getGreeting() {

return greetingService.getGreetingMessage();

}

}

**Step 5**: Open the main class and put the below code.

package org.example.springpropertiesdemo;

​

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

​

@SpringBootApplication

public class SpringPropertiesDemoApplication {

​

public static void main(String[] args) {

SpringApplication.run(SpringPropertiesDemoApplication.class, args);

}

​

}

**pom.xml:**

<?xml version="1.0" encoding="UTF-8"?>

**<project** xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd"**>**

**<modelVersion>**4.0.0**</modelVersion>**

**<parent>**

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

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

**<version>**3.2.4**</version>**

**<relativePath/>** *<!-- lookup parent from repository -->*

**</parent>**

**<groupId>**org.example**</groupId>**

**<artifactId>**spring-properties-demo**</artifactId>**

**<version>**0.0.1-SNAPSHOT**</version>**

**<name>**spring-properties-demo**</name>**

**<description>**spring-properties-demo**</description>**

**<properties>**

**<java.version>**17**</java.version>**

**</properties>**

**<dependencies>**

**<dependency>**

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

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

**</dependency>**

**<dependency>**

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

**<artifactId>**spring-boot-devtools**</artifactId>**

**<scope>**runtime**</scope>**

**<optional>**true**</optional>**

**</dependency>**

**<dependency>**

**<groupId>**org.projectlombok**</groupId>**

**<artifactId>**lombok**</artifactId>**

**<optional>**true**</optional>**

**</dependency>**

**<dependency>**

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

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

**<scope>**test**</scope>**

**</dependency>**

**</dependencies>**

**<build>**

**<plugins>**

**<plugin>**

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

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

**<configuration>**

**<excludes>**

**<exclude>**

**<groupId>**org.projectlombok**</groupId>**

**<artifactId>**lombok**</artifactId>**

**</exclude>**

**</excludes>**

**</configuration>**

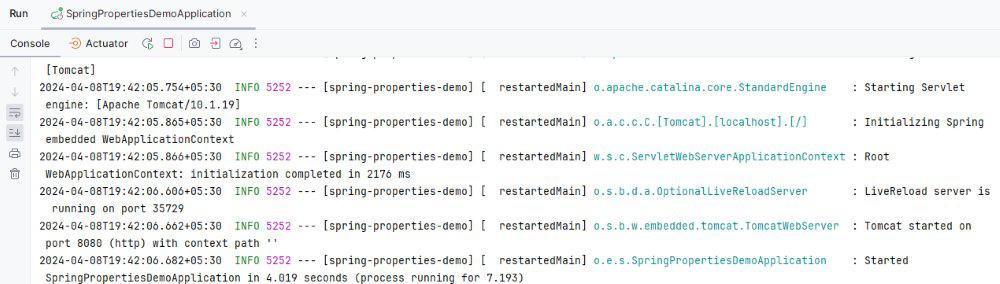
**</plugin>**

**</plugins>**

**</build>**

**</project>**

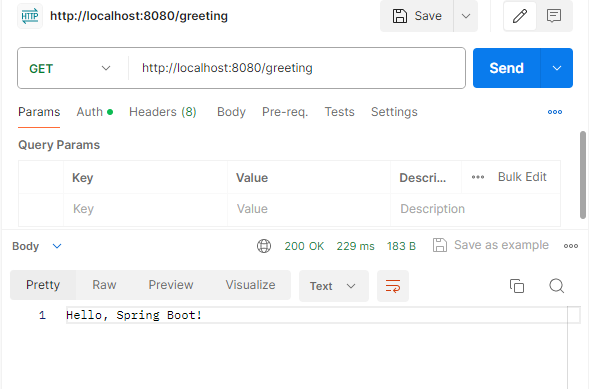
**Step 6**: Once complete the spring project and it run as spring application once it runs successful then it starts at port 8080.



API Test:

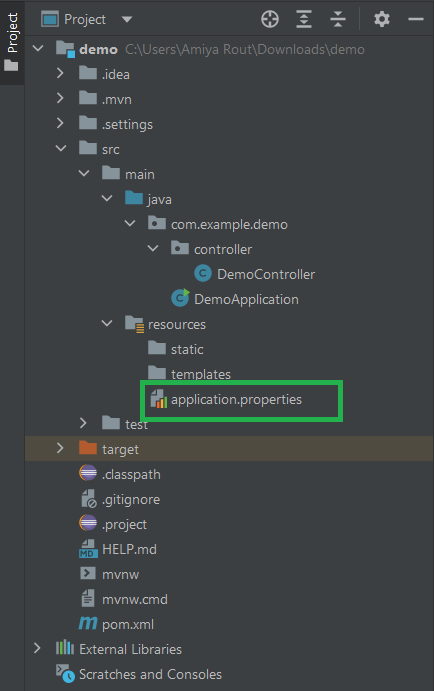
GET http://localhost:8080/greeting

**Image Output:**



**5.Spring Boot - application.yml/application.yaml File.**

Spring is widely used for creating scalable applications. For web applications Spring provides. In [Spring Boot](https://www.geeksforgeeks.org/springboot/introduction-to-spring-boot/), whenever we create a new Spring Boot Application in spring starter, or inside an IDE (Eclipse or STS) a file is located inside the ***src/main/resources*** folder named as **application.properties**file which is shown in the below media:



So in a spring boot application, **application.properties** file is used to write the application-related property into that file. This file contains the different configuration which is required to run the application in a different environment, and each environment will have a different property defined by it. Inside the application properties file, we define every type of property like changing the port, database connectivity, connection to the eureka server, and many more. But sometimes there is another file is located inside the ***src/main/resources*** folder named as **application.yml/application.yaml**file and the code present inside this file is present in a hierarchical format which is shown in the below image:

A screenshot of a computer program

AI-generated content may be incorrect.

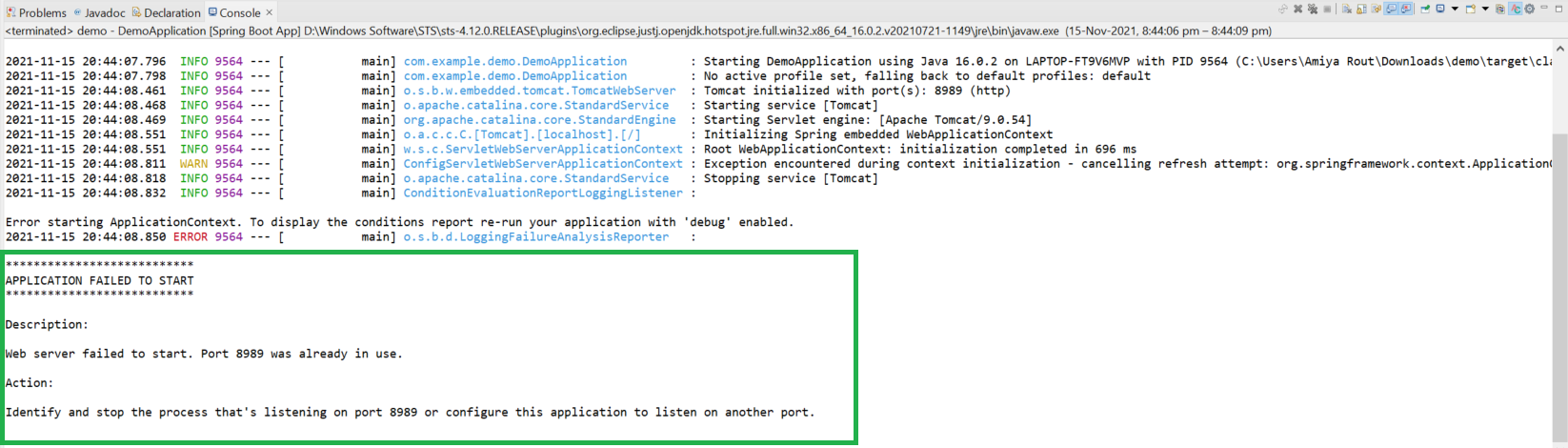
So what's this **application.yml**file? **YAML stands for Yet Another Markup Language or YAML ain't markup language (a recursive acronym), which emphasizes that YAML is for data, not documents.**YAML is a data serialization language that is often used for writing configuration files. So YAML configuration file in Spring Boot provides a very convenient syntax for storing logging configurations in a hierarchical format. The **application.properties** file is not that readable. So most of the time developers choose application.yml file over application.properties file. YAML is a superset of JSON, and as such is a very convenient format for specifying hierarchical configuration data. YAML is more readable and it is good for the developers to read/write configuration files.

Now let's see some examples for better understanding via proposing different examples as listed and described later as follows:

1. To Change the Port Number
2. To define the name of our application
3. Connecting with the MySQL Database
4. Connecting with the H2 Database
5. Connecting with the MongoDB Database
6. Connecting with the Eureka Server

**Example 1:**To Change the Port Number

Sometimes when you run your spring application you may encounter the following type of error



The error is Port 8989 was already in use. So in this case you may kill that process that is running on this port number or you may change your port number and rerun your application. So where do you have to change your port number? e.g in the **application.properties**file or in**application.yml**file. So you can change your port number by the following line

server:

port:

8082

**Example 2:**To define the name of our application

To define the name of our application you can write the properties like this

spring:

application:

name: userservice

So you can see this represents the property as key-value pair here, every key associated with a value also.

**Example 3:**Connecting with the MySQL Database

To connect with the MySQL Database you have to write a bunch of lines. You can write the properties like this

spring:

datasource:

driver-class-name: com.mysql.jdbc.Driver

username: springuser

url: jdbc:mysql://${MYSQL\_HOST:localhost}:3306/db\_example

password: ThePassword

jpa:

hibernate:

ddl-auto: update

**Example 4:**Connecting with the H2 Database

H2 is an embedded, open-source, and in-memory database. It is a relational database management system written in Java. It is a client/server application. It is generally used in unit testing. It stores data in memory, not persist the data on disk. To connect with the H2 Database you have to write a bunch of lines. You can write the properties like this

spring:

h2:

console:

enabled: 'true'

datasource:

username: sa

url: jdbc:h2:mem:dcbapp

driverClassName: org.h2.Driver

password: password

jpa:

database-platform: org.hibernate.dialect.H2Dialect

**Example 5:**Connecting with the MongoDB Database

To connect with the MongoDB Database you have to write a bunch of lines. You can write the properties like this

spring:

data:

mongodb:

database: BookStore

port: '27017'

host: localhost

**Example 6:**Connecting with the Eureka Server

Eureka Server is an application that holds information about all client-service applications. Every Microservice will register into the Eureka server and the Eureka server knows all the client applications running on each port and IP address. Eureka Server is also known as Discovery Server. You can write the properties like this

eureka:

client:

service-url:

defaultZone: http://localhost:9096/eureka/

fetch-registry: 'true'

register-with-eureka: 'true'

instance:

hostname: localhost

***Note****: The value written here is sample data. Please write the values as per your requirements. But the keys remain the same.*

6.Method-Level Annotations.

Method-level annotations are applied directly to individual methods within a class. They provide specific instructions or configurations for that particular method.

Examples:

* @Bean: Used within a @Configuration class to declare a method that produces a Spring-managed bean. The return value of the method becomes the bean registered in the Spring application context.

Java

@Configuration  
 public class AppConfig {  
 @Bean  
 public MyService myService() {  
 return new MyServiceImpl();  
 }  
 }

* @RequestMapping (and its specialized variants like @GetMapping, @PostMapping, etc.): Used in controller classes to map incoming HTTP requests to specific handler methods.

Java

@RestController  
 public class MyController {  
 @GetMapping("/hello")  
 public String sayHello() {  
 return "Hello, Spring!";  
 }  
 }

* @Transactional: Applied to a method to indicate that the method should be executed within a transactional context. Spring will manage the transaction, committing it on success or rolling it back on exceptions.

Java

@Service  
 public class UserService {  
 @Transactional  
 public void createUser(User user) {  
 *// Logic to create user in the database*  
 }  
 }

* @Secured / @PreAuthorize / @PostAuthorize: Used in Spring Security to define access control rules for specific methods based on roles or expressions.

Java

@Service  
 public class AdminService {  
 @Secured("ROLE\_ADMIN")  
 public void deleteUser(Long userId) {  
 *// Logic to delete user*  
 }  
 }

Type-Level Annotations (Class-Level Annotations)

Type-level annotations are applied to entire classes, providing configurations or metadata that apply to all methods and fields within that class (unless overridden by method-level annotations).

Examples:

* @Configuration: Marks a class as a source of bean definitions. It's often used in conjunction with @Bean methods.

Java

@Configuration  
 public class AppConfig {  
 *// ... @Bean methods*  
 }

* @Controller / @RestController: Designates a class as a Spring MVC controller, handling web requests. @RestController is a convenience annotation combining @Controller and @ResponseBody.

Java

@RestController  
 public class MyController {  
 *// ... request mapping methods*  
 }

* @Service: Marks a class as a service component in the business layer, typically containing business logic.

Java

@Service  
 public class MyServiceImpl implements MyService {  
 *// ... service methods*  
 }

* @Repository: Indicates that a class is a Data Access Object (DAO) or repository, providing an abstraction over data storage.

Java

@Repository  
 public class UserRepository {  
 *// ... data access methods*  
 }

* @Component: A generic stereotype annotation indicating that a class is a Spring-managed component. @Controller, @Service, and @Repository are specialized forms of @Component.

Java

@Component  
 public class UtilityClass {  
 *// ... utility methods*  
 }

* @SpringBootApplication: A meta-annotation that combines @SpringBootConfiguration, @EnableAutoConfiguration, and @ComponentScan. It's typically used on the main application class to bootstrap a Spring Boot application.

Java

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