Spring - Stereotype Annotations

Last Updated : 23 Jul, 2025

Spring is one of the most popular Java EE frameworks. It is an open-source lightweight framework that allows Java EE 7 developers to build simple, reliable, and scalable enterprise applications. This framework mainly focuses on providing various ways to help you manage your business objects.

Spring Annotations are a form of metadata that provides data about a program. Annotations are used to provide supplemental information about a program. They do not have a direct effect on the operation of the code they annotate, nor do they change the action of the compiled program.

Stereotype Annotations

Spring Framework provides us with some special annotations. These annotations are used to create Spring beans automatically in the application context. **@Component annotation is the main Stereotype Annotation**.

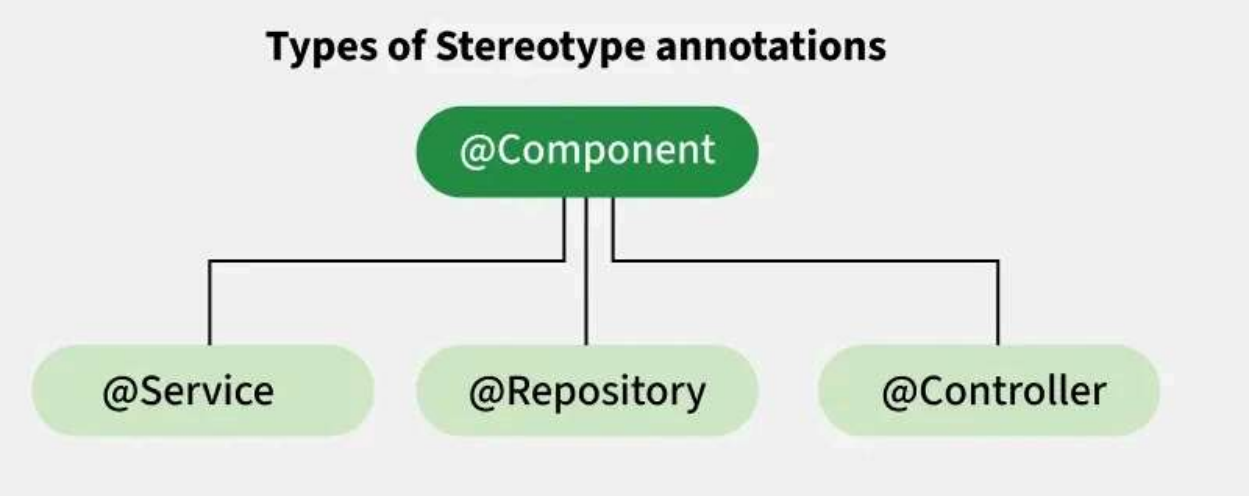
Note: @Component annotation is a generic stereotype for any Spring-managed component.

There are some Stereotype meta-annotations, which are derived from @Component:

- **@Service:** This annotation is used to indicate that a class holds business logic.
- **@Repository:** This annotation is used to indicate that a class deals with CRUD operations, typically used with DAO or Repository implementations.

- **@Controller:** This annotation is used to indicate that a class is a front controller, responsible for handling user requests.

So the stereotype annotations in spring are @Component, @Service, @Repository, and @Controller.



@Component Annotation

@Component is a class-level annotation. It is used to denote a class as a Component. We can use @Component across the application to mark the beans as Spring's managed components. A component is responsible for some operations.

Implementation of @Component Annotation in Spring Boot

Let's create a very simple Spring boot application to showcase the use of Spring Component annotation and how Spring autodetects it with annotation-based configuration and classpath scanning.

Step 1: Create a Simple Spring Boot Project. Geek, you need pre-requisite of creating and setting up Spring Boot Project

Refer to this article Create and Setup Spring Boot Project in Eclipse IDE and create a simple spring boot project.

Step 2: Add the spring-context dependency in your pom.xml file. Go to the pom.xml file inside your project and add the following spring-context dependency.

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.3.13</version>
</dependency>
```

Step 3: Create a simple component class.

Go to the **src > main > java > your package name > right-click > New > Java Class** and create your component class and mark it with **@Component** annotation.

```
// Java Program to Illustrate Component class
package com.example.demo;

import org.springframework.stereotype.Component;

// Annotation
@Component

// Class
public class ComponentDemo {

    // Method
    public void demoFunction()
    {

        // Print statement when method is called
        System.out.println("Hello GeeksForGeeks");
    }
}
```

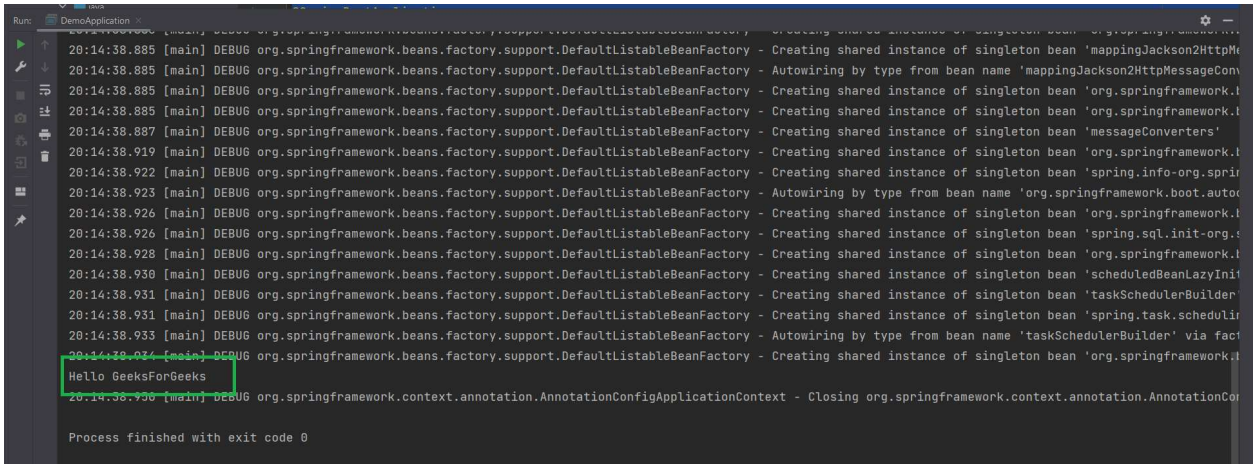
Step 4: Create an annotation-based spring context

Now go to your Application (@SpringBootApplication) file and here in this file create an annotation-based spring context and get the ComponentDemo bean from it.

```
// Java Program to Illustrate Application class

// Importing package here
```

```
package com.example.demo;
// Importing required classes
import org.springframework.boot.autoconfigure.SpringBootApplication;
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;

// Class
public class DemoApplication {

    // Main driver method
    public static void main(String[] args)
    {

        // Annotation based spring context
        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext();
        context.scan("com.example.demo");
        context.refresh();

        // Getting the Bean from the component class
        ComponentDemo componentDemo = context.getBean(ComponentDemo.class);
        componentDemo.demoFunction();

        // Closing the context
        // using close() method
        context.close();
    }
}
```

Output:



So you can see the power of @Component annotation, we didn't have to do anything to inject our component to spring context.

@Service Annotation

In an application, the business logic resides within the service layer so we use the **@Service Annotation** to indicate that a class belongs to that layer.

It is also a specialization of **@Component Annotation** like the **@Repository Annotation**. One most important thing about the @Service Annotation is it can be applied only to classes. It is used to mark the class as a service provider. So overall @Service annotation is used with classes that provide some business functionalities. Spring context will autodetect these classes when annotation-based configuration and classpath scanning is used.

Implementation of @Service Annotation in Spring Boot

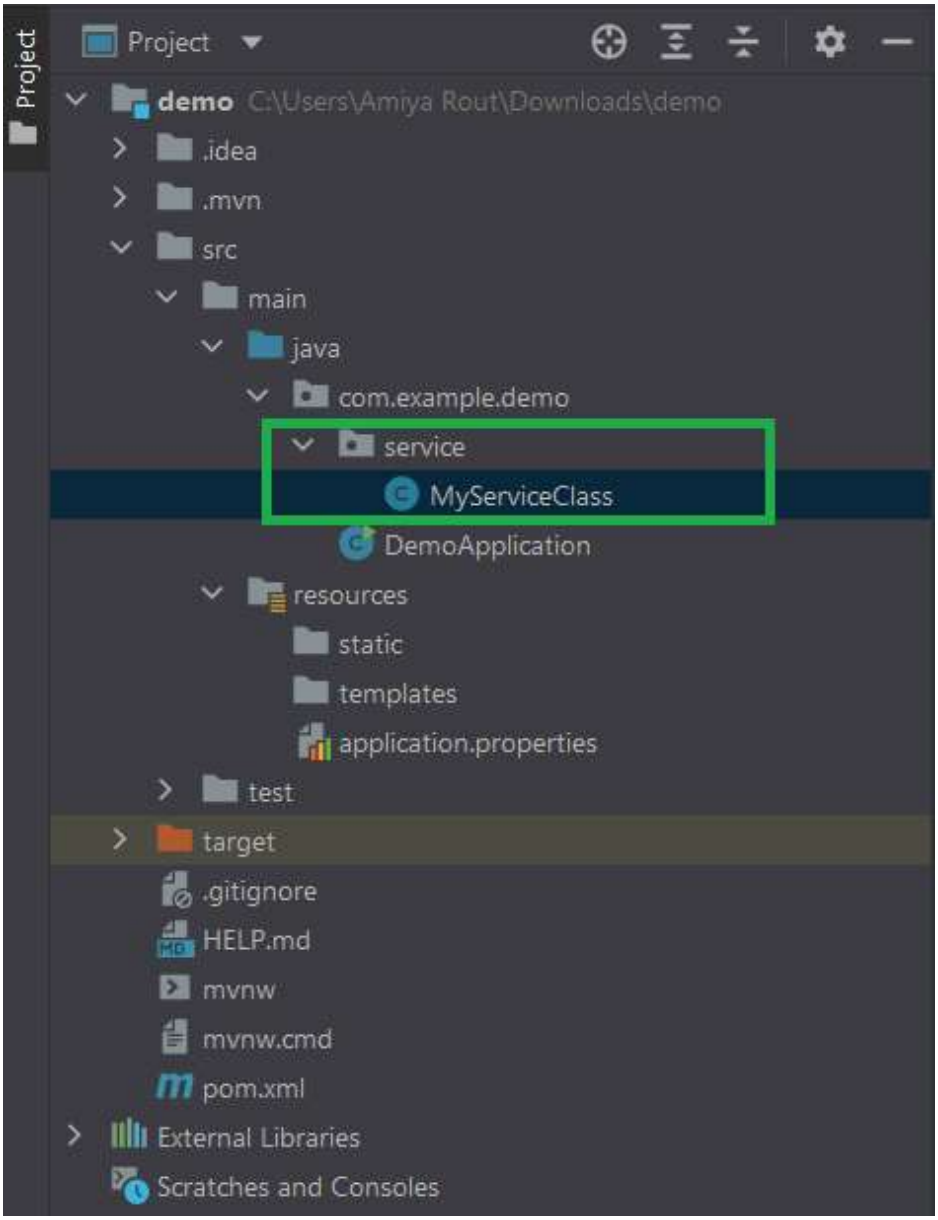
Step 1: Create a Simple Spring Boot Project

Refer to this article Create and Setup Spring Boot Project in Eclipse IDE and create a simple spring boot project.

Step 2: Add the spring-context dependency in your pom.xml file. Go to the pom.xml file inside your project and add the following spring-context dependency.

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.3.13</version>
</dependency>
```

Step 3: In your project create one package and name the package as "service". In the service package create a class and name it as **MyServiceClass**. This is going to be our final project structure.

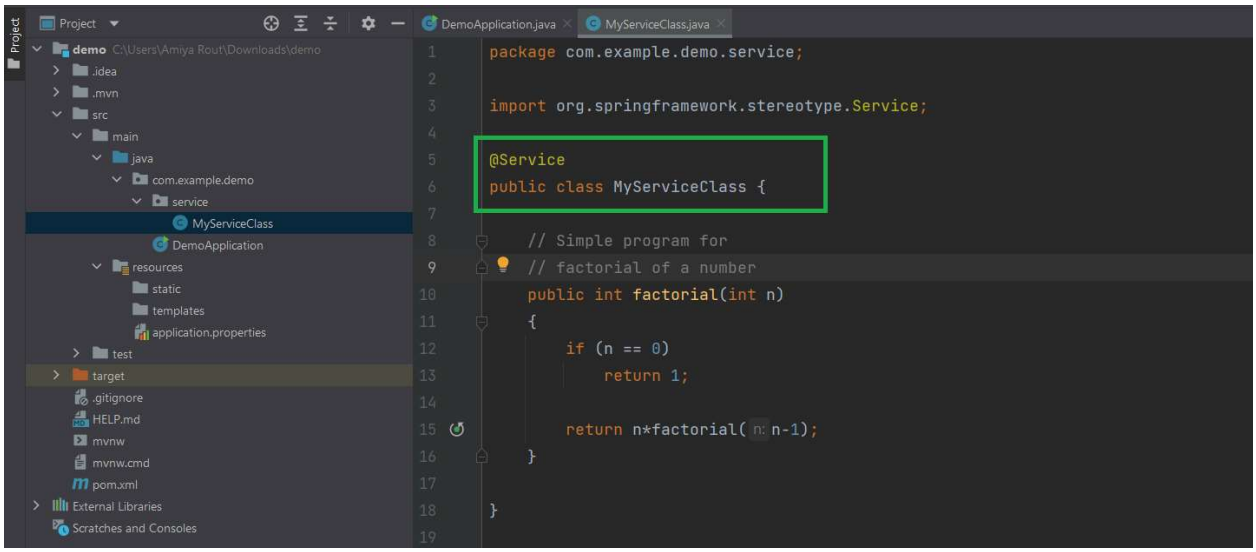


Below is the code for the **MyServiceClass.java** file.

```
package com.example.demo.service;

import org.springframework.stereotype.Service;

@Service
public class MyServiceClass {

    // Simple program for
    // factorial of a number
    public int factorial(int n)
    {
        if (n == 0)
            return 1;

        return n*factorial(n-1);
    }
```

```
}
```

In this code notice that it's a simple java class that provides functionalities to calculate the factorial of a number. So we can call it a service provider. We have annotated it with @Service annotation so that spring-context can autodetect it and we can get its instance from the context.



Step 4: Spring Repository Test

So now our Spring Repository is ready, let's test it out. Go to the **DemoApplication.java** file and refer to the below code.

```
package com.example.demo;

import com.example.demo.service.MyServiceClass;
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class DemoApplication {

    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext("com.example.demo");
        MyServiceClass myServiceClass =
context.getBean(MyServiceClass.class);
        int factorialOf5 = myServiceClass.factorial(5);
        System.out.println("Factorial of 5 is: " + factorialOf5);
        context.close();
    }
}
```

Output:

@Repository Annotation

@Repository Annotation is a specialization of **@Component** annotation which is used to indicate that the class provides the mechanism for storage, retrieval, update, delete and search operation on objects. Though it is a specialization of @Component annotation, so Spring Repository classes are autodetected by spring framework through classpath scanning. This annotation is a general-purpose stereotype annotation which very close to the DAO pattern where DAO classes are responsible for providing CRUD operations on database tables.

Implementation of @Repository Annotation in Spring Boot

Step 1: Create a Simple Spring Boot Project

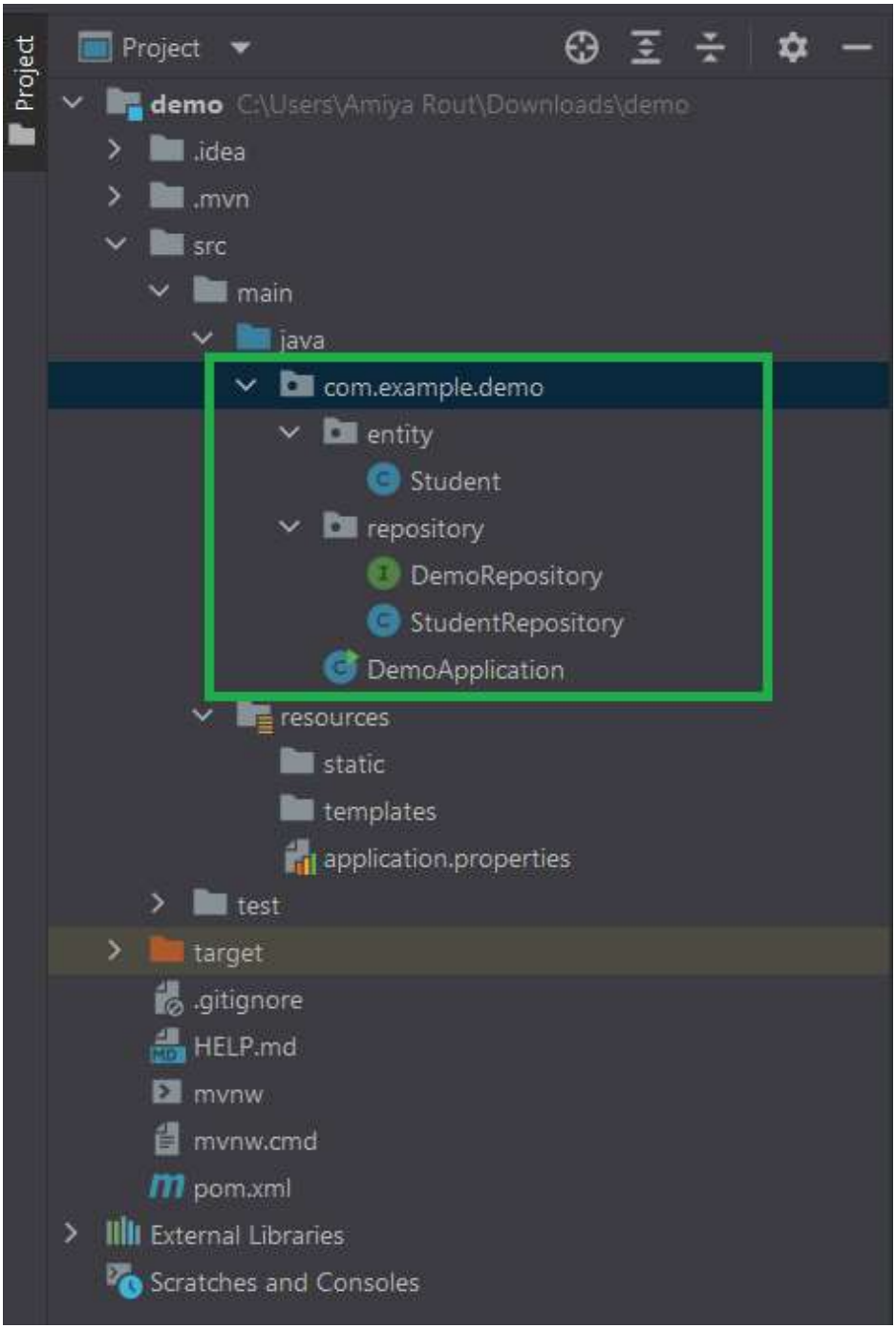
Refer to this article Create and Setup Spring Boot Project in Eclipse IDE and create a simple spring boot project.

Step 2: Add the spring-context dependency in your pom.xml file. Go to the pom.xml file inside your project and add the following spring-context dependency.

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
```

```
    <version>5.3.13</version>
</dependency>
```

Step 3: In your project create two packages and name the package as "entity" and "repository". In the entity package create a class name it as Student. In the repository, package create a Generic Interface name it as DemoRepository and a class name it as StudentRepository. This is going to be our final project structure.



Step 4: Create an entity class for which we will implement a spring repository. Here our entity class is Student. Below is the code for the **Student.java** file. This is a simple **POJO** (Plain Old Java Object) class in java.

```
package com.example.demo.entity;

public class Student {

    private Long id;
    private String name;
    private int age;

    public Student(Long id, String name, int age) {
        this.id = id;
        this.name = name;
        this.age = age;
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return "Student{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", age=" + age +
                '}';
```

```
    }
}
```

Step 5: Before implementing the Repository class we have created a generic DemoRepository interface to provide the contract for our repository class to implement. Below is the code for the **DemoRepository.java** file.

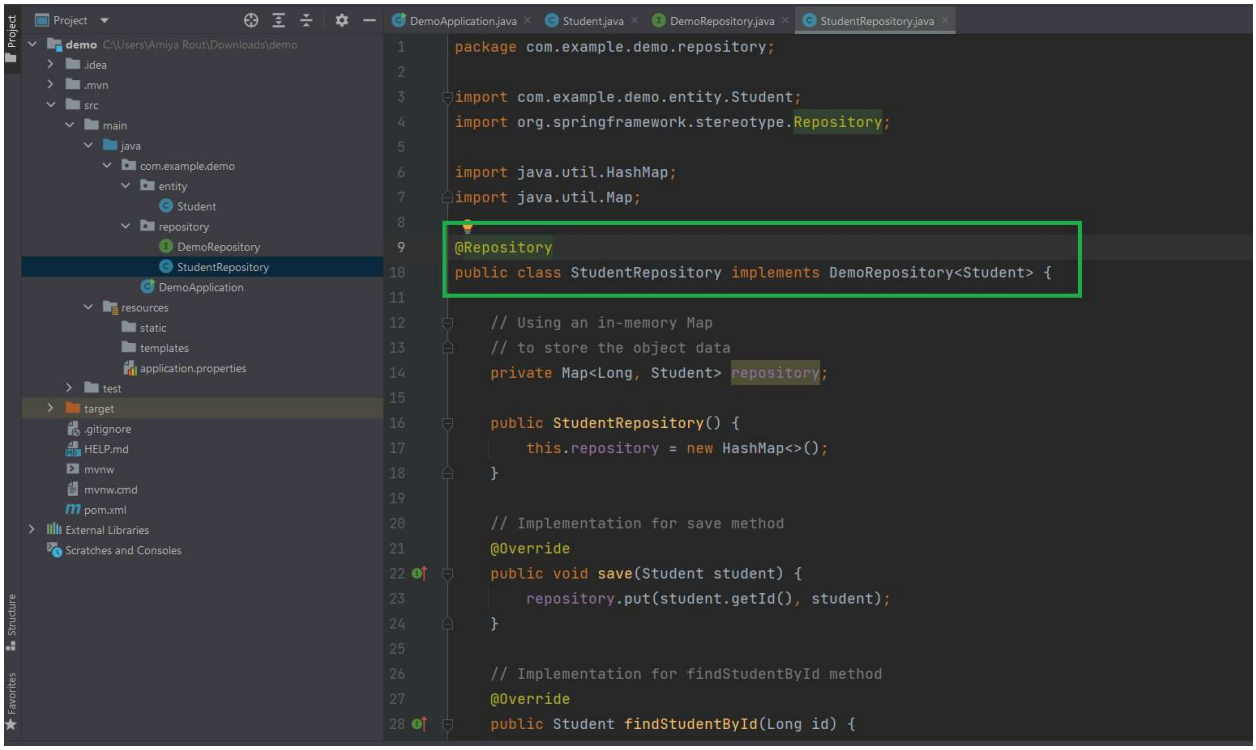
```
// Java Program to illustrate DemoRepository File

package com.example.demo.repository;

public interface DemoRepository<T> {

    // Save method
    public void save(T t);

    // Find a student by its id
    public T findStudentById(Long id);

}
```

Step 6: Now let's look at our StudentRepository class implementation.

```
// Java Program to illustrate StudentRepository File

package com.example.demo.repository;

import com.example.demo.entity.Student;
import org.springframework.stereotype.Repository;

import java.util.HashMap;
import java.util.Map;

@Repository
public class StudentRepository implements DemoRepository<Student> {

    // Using an in-memory Map
    // to store the object data
    private Map<Long, Student> repository;

    public StudentRepository() {
        this.repository = new HashMap<>();
    }

    // Implementation for save method
    @Override
```

```
    public void save(Student student) {
        repository.put(student.getId(), student);
    }

    // Implementation for findStudentById method
    @Override
    public Student findStudentById(Long id) {
        return repository.get(id);
    }
}
```

In this **StudentRepository.java** file, you can notice that we have added the @Repository annotation to indicate that the class provides the mechanism for storage, retrieval, update, delete and search operation on objects.



> ***Note**: Here we have used an in-memory Map to store the object data, you can use any other mechanisms too. In the real world, we use Databases to store object data.*

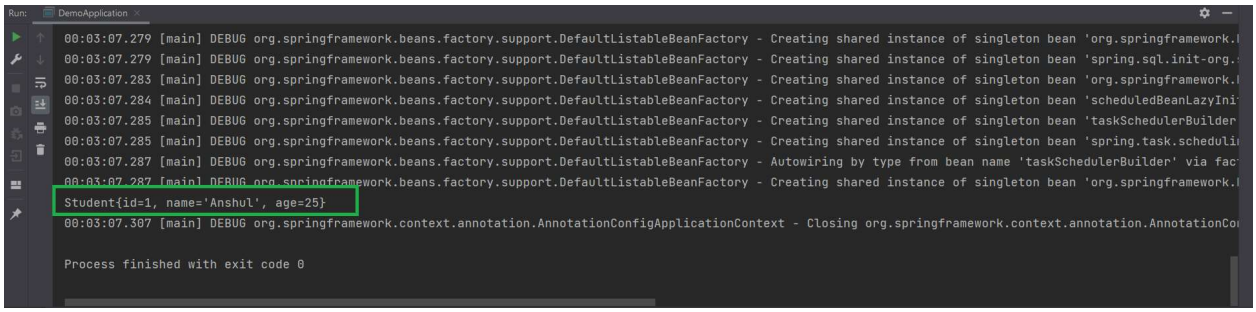
Step 7: Spring Repository Test

So, now our Spring Repository is ready, let's test it out. Go to the **DemoApplication.java** file and refer to the below code.

```
package com.example.demo;

import com.example.demo.entity.Student;
import com.example.demo.repository.StudentRepository;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class DemoApplication {

    public static void main(String[] args) {

        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext("com.example.demo");

        StudentRepository repository =
context.getBean(StudentRepository.class);

        // testing the store method
        repository.save(new Student(1L, "Anshul", 25));
        repository.save(new Student(2L, "Mayank", 23));

        // testing the retrieve method
        Student student = repository.findStudentById(1L);
        System.out.println(student);

        // close the spring context
        context.close();
    }

}
```

Output:

Lastly, run your application and you should get the following output as shown below as follows:



@Controller Annotation

Spring @Controller annotation is also a specialization of **@Component** annotation. The @Controller annotation indicates that a particular class serves the role of a **controller**. Spring Controller annotation is typically used in combination with annotated handler methods based on the **@RequestMapping** annotation. It can be applied to classes only. It's used to mark a class as a web request handler. It's mostly used with Spring MVC applications. This annotation acts as a stereotype for the annotated class, indicating its role. The dispatcher scans such annotated classes for mapped methods and detects **@RequestMapping** annotations. Let's understand all of these by example.

Implementation of @Controller Annotation in Spring Boot

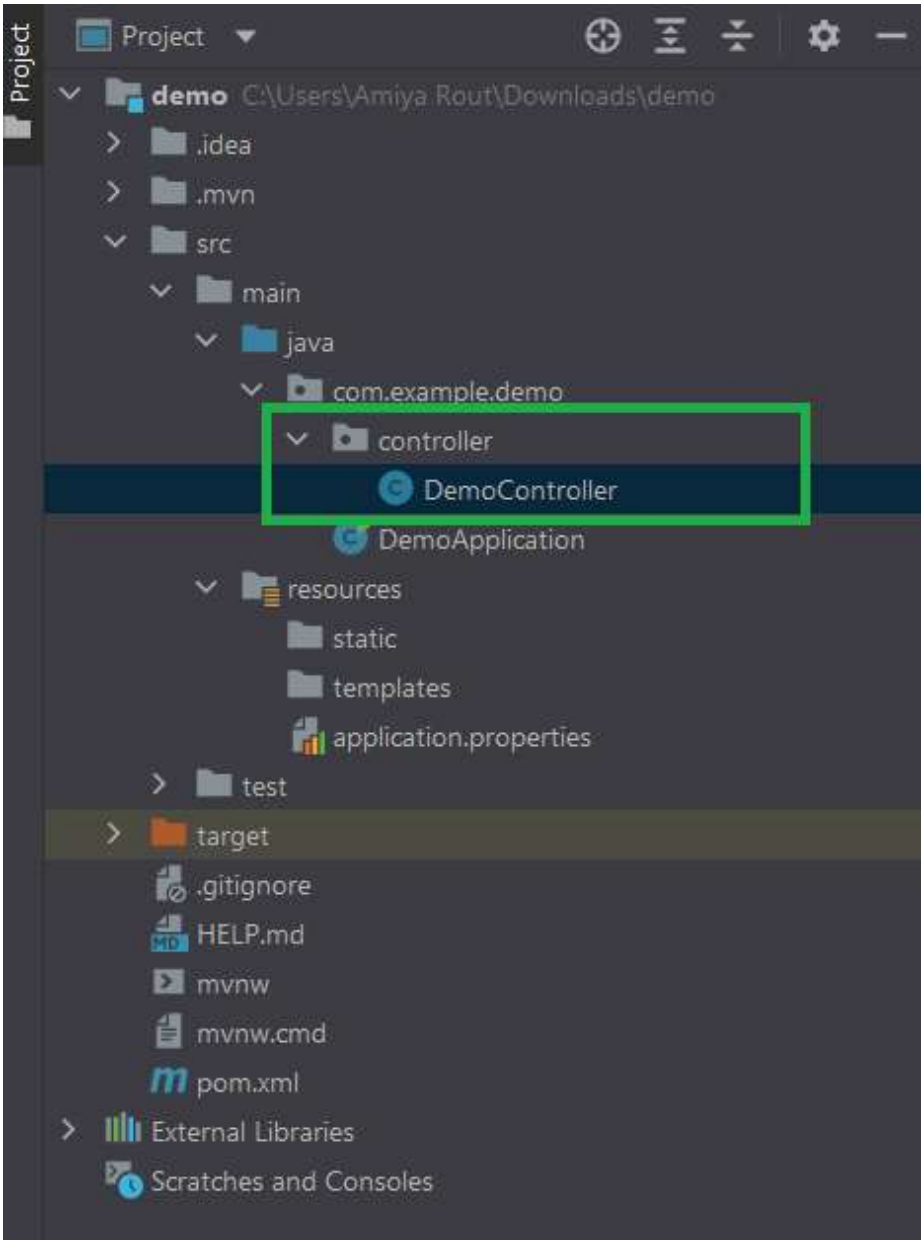
Step 1: Create a Simple Spring Boot Project

Refer to this article Create and Setup Spring Boot Project in Eclipse IDE and create a simple spring boot project.

Step 2: Add the spring-web dependency in your pom.xml file. Go to the pom.xml file inside your project and add the following spring-web dependency.

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

Step 3: In your project create one package and name the package as "controller". In the controller package create a class and name it as **DemoController**. This is going to be our final project structure.



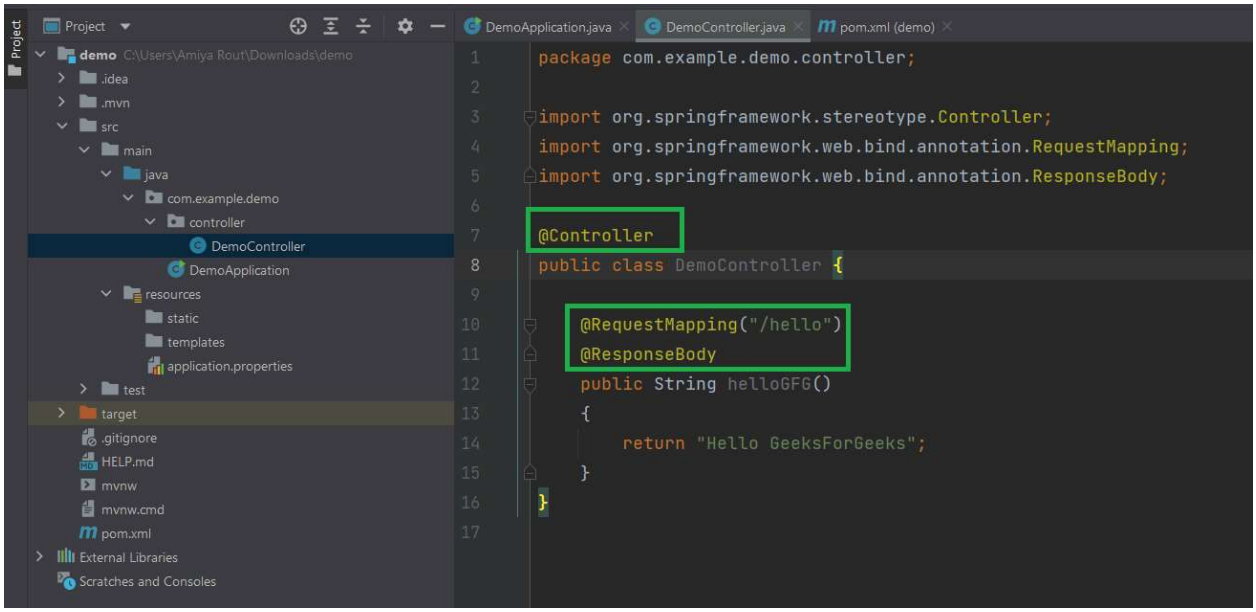
Below is the code for the **DemoController.java** file.

```
package com.example.demo.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class DemoController {

    @RequestMapping("/hello")
    @ResponseBody
    public String helloGFG()
    {
        return "Hello GeeksForGeeks";
```

```
    }
}
```

We have used the below annotations in our controller layer. Here in this example, the URI path is ***/hello***.

- **@Controller:** This is used to specify the controller.
- **@RequestMapping:** This is used to map to the Spring MVC controller method.
- **@ResponseBody:** Used to bind the HTTP response body with a domain object in the return type.

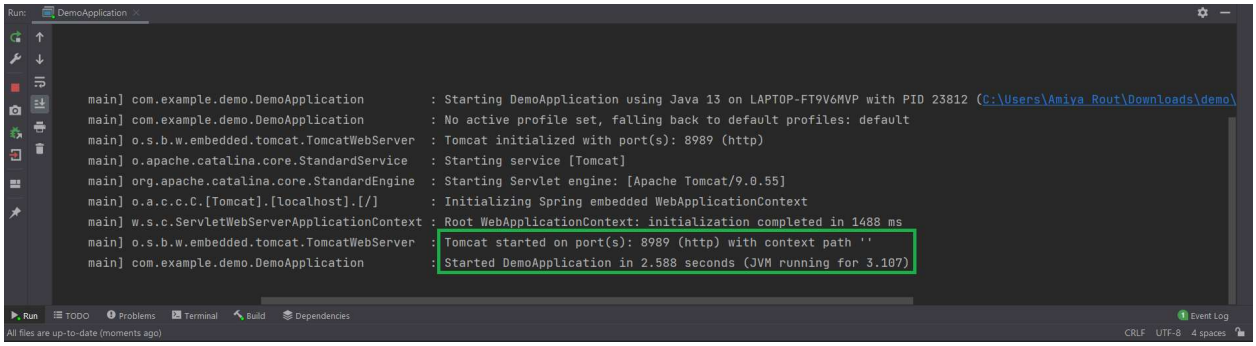


Step 4: Run the application. There is no need to change anything inside the **DemoApplication.java** file.

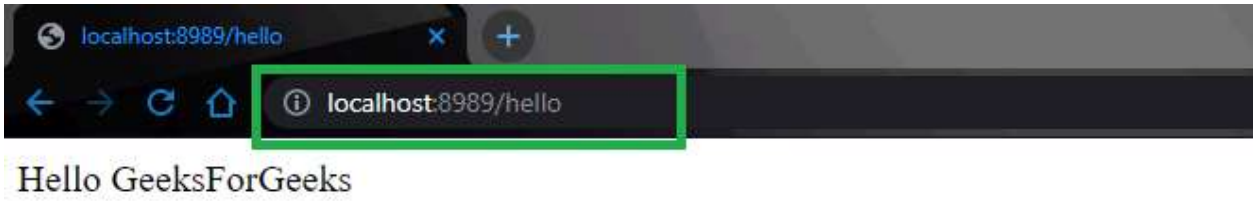
```
package com.example.demo;

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

@SpringBootApplication
public class DemoApplication {

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

```
}
```

Output:



Try this Tomcat URL, which is running on ***http://localhost:8989/hello***



Comment | More info | Advertise with us

GeeksforGeeks
Sanchhaya Education Private Limited
Corporate & Communications Address:
A-143, 7th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh (201305)