Experiment 1 Spring Framework Annotations

Objective: Students will be able to implement spring boot application using Spring Framework Annotations

Theory:

Spring framework 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. Spring 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. In this article, we will discuss the main types of annotations available in the Spring Framework with examples.

Types of Spring Framework Annotations

- Spring Core Annotations
- Spring Web Annotations
- Spring Boot Annotations
- Spring Scheduling Annotations
- Spring Data Annotations
- Spring Bean Annotations

The below diagram demonstrates the types of Spring Framework Annotations

Spring Bean Annotations
06
01
Spring Core Annotations
Spring Framework Annotations
Spring Data Annotations
05
02
Spring Web Annotations
Spring Scheduling Annotations
04
03
Spring Boot Annotations

Spring Boot Annotations

Spring Boot annotations are present in the **org.springframework.boot.autoconfigure** and **org.springframework.boot.autoconfigure.condition** packages. Some of the commonly used annotations in this category are:

@SpringBootApplication: This annotation is used to mark the main class of a Spring Boot application. It encapsulates @Configuration, @EnableAutoConfiguration, and @ComponentScan annotations with their default attributes.

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

- **@EnableAutoConfiguration:** This annotation enables Spring Boot's auto-configuration mechanism.
- **@ConditionalOnClass:** This annotation configures a bean only if a specified class is present on the classpath.
- **@ConditionalOnMissingBean:** This annotation configures a bean only if a specified bean is not already present in the application context.

Step-by-Step Guide to Create a Spring Boot Project in Eclipse

1. Install Eclipse IDE

If you don't already have Eclipse, you can download and install the **Eclipse IDE for Java Developers**

Make sure you have JDK installed as well (JDK 8 or higher).

2. Install Spring Tools (Spring Tool Suite)

Spring Tools (STS) is a set of Eclipse plugins that helps in developing Spring-based applications.

Steps:

- Go to **Eclipse IDE**.
- Click on `Help` -> `Eclipse Marketplace`.
- In the **Eclipse Marketplace**, search for **Spring Tools** or **Spring Tool Suite**.
- Install **Spring Tools 4** and restart Eclipse.

3. Create a New Spring Boot Project

Steps:

1. Open Eclipse and go to the **File** menu.
2. Click on **New** -> **Other...**.
3. In the **New Project** dialog, search for **Spring Starter Project** and select it.
4. Click **Next**.

Fill in the project details:

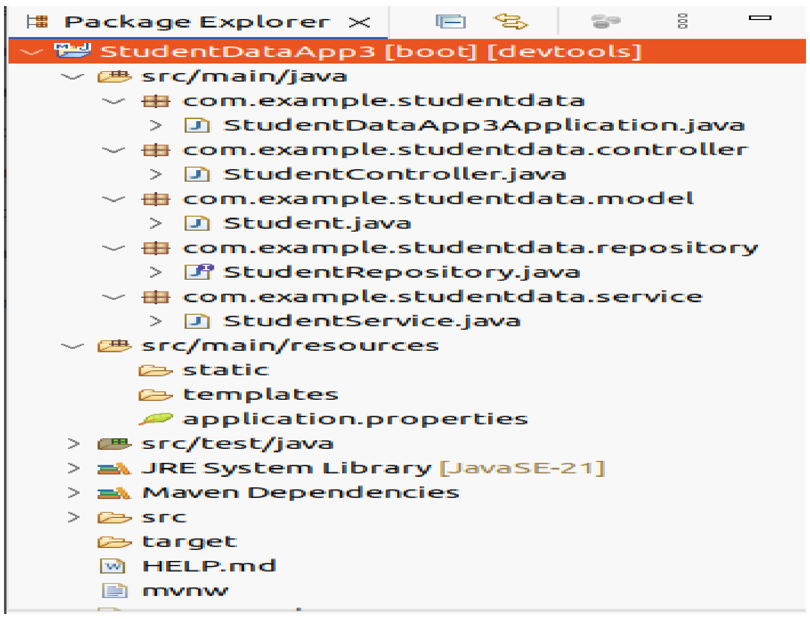
- **Name**: `StudentDataApp`
- **Type**: `Maven Project`
- **Packaging**: `Jar`
- **Java Version**: Select the appropriate JDK version (JDK 8 or higher).
- **Dependencies**: Choose the following dependencies:
    - `Spring Web`
    - `Spring Data JPA`
    - `H2 Database` (For an in-memory database)
    - `Spring Boot DevTools` (optional, for live reload)
    - `Spring Boot Starter Validation` (optional, for validation)

Click **Finish** after filling in the details.

---

4. Set Up the Project Structure

Once the project is created, you should see a structure similar to the following:



```
Package Explorer
StudentDataApp3 [boot] [devtools]
  src/main/java
    com.example.studentdata
      > StudentDataApp3Application.java
    com.example.studentdata.controller
      > StudentController.java
    com.example.studentdata.model
      > Student.java
    com.example.studentdata.repository
      > StudentRepository.java
    com.example.studentdata.service
      > StudentService.java
  src/main/resources
    static
    templates
    application.properties
> src/test/java
> JRE System Library [JavaSE-21]
> Maven Dependencies
> src
  target
  HELP.md
  mvnw
```

5. Create the Student Model (Entity)

Go to `src/main/java/com/example/studentdata/model` and create a class `Student.java`:

package com.example.studentdata.model;

import javax.persistence.Entity;
import javax.persistence.Id;

```
@Entity
public class Student {

    @Id
    private Long id;
    private String name;
    private String email;
    private String course;

    // Default constructor
    public Student() {}

    // Constructor with fields
    public Student(Long id, String name, String email, String course) {
        this.id = id;
        this.name = name;
        this.email = email;
        this.course = course;
    }

    // Getters and Setters
    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 String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getCourse() {
        return course;
    }

    public void setCourse(String course) {
        this.course = course;
    }
}
```

6. Create the Student Repository

Go to `src/main/java/com/example/studentdata/repository` and create an interface `StudentRepository.java`:

```
package com.example.studentdata.repository;

import com.example.studentdata.model.Student;
import org.springframework.data.jpa.repository.JpaRepository;

public interface StudentRepository extends JpaRepository<Student, Long> {
   // You can define custom queries if needed
}
```

7. Create the Student Service

Go to `src/main/java/com/example/studentdata/service` and create a class `StudentService.java`:

```
package com.example.studentdata.service;

import com.example.studentdata.model.Student;
import com.example.studentdata.repository.StudentRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.Optional;

@Service
public class StudentService {

   @Autowired
   private StudentRepository studentRepository;

   // Save or update student
   public Student saveStudent(Student student) {
      return studentRepository.save(student);
   }

   // Get all students
   public List<Student> getAllStudents() {
      return studentRepository.findAll();
   }

   // Get student by ID
   public Optional<Student> getStudentById(Long id) {
      return studentRepository.findById(id);
   }
```

```
    // Delete student by ID
    public void deleteStudent(Long id) {
        studentRepository.deleteById(id);
    }
}
```

8. Create the Student Controller

Go to `src/main/java/com/example/studentdata/controller` and create a class `StudentController.java`:

```
package com.example.studentdata.controller;

import com.example.studentdata.model.Student;
import com.example.studentdata.service.StudentService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;
import java.util.Optional;

@RestController
@RequestMapping("/students")
public class StudentController {

    @Autowired
    private StudentService studentService;

    // Create or update a student
    @PostMapping
    public Student createStudent(@RequestBody Student student) {
        return studentService.saveStudent(student);
    }

    // Get all students
    @GetMapping
    public List<Student> getAllStudents() {
        return studentService.getAllStudents();
    }

    // Get student by ID
    @GetMapping("/{id}")
    public Optional<Student> getStudentById(@PathVariable Long id) {
        return studentService.getStudentById(id);
    }

    // Delete student by ID
    @DeleteMapping("/{id}")
    public void deleteStudent(@PathVariable Long id) {
        studentService.deleteStudent(id);
    }
}
```

9. Configure Database (Optional)

If you are using an **in-memory H2 database**, you can configure it in `src/main/resources/application.properties`:

```
# H2 Database Configuration
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=update
spring.h2.console.enabled=true
```

This configuration sets up an in-memory H2 database for development and testing.

---

10. Run the Application

- In Eclipse, right-click on the `StudentDataApplication.java` file (located under `src/main/java/com/example/studentdata`).
- Select **Run As** -> **Java Application**.

Spring Boot will start an embedded Tomcat server, and your application will be accessible at `http://localhost:8080`

11. Testing the API

Use Postman or curl to test the APIs.

- **POST** to `http://localhost:8080/students` to create a student (with JSON data like):

```
{
  "id": 1,
  "name": "John Doe",
  "email": "john.doe@example.com",
  "course": "Computer Science"
}
```

GET to `http://localhost:8080/students` to retrieve all students.

- **GET** to `http://localhost:8080/students/{id}` to get a student by ID.
- **DELETE** to `http://localhost:8080/students/{id}` to delete a student.

12. Shut Down the Application

When you’re done, you can stop the application by clicking the **Stop** button in Eclipse.

Pom.xml

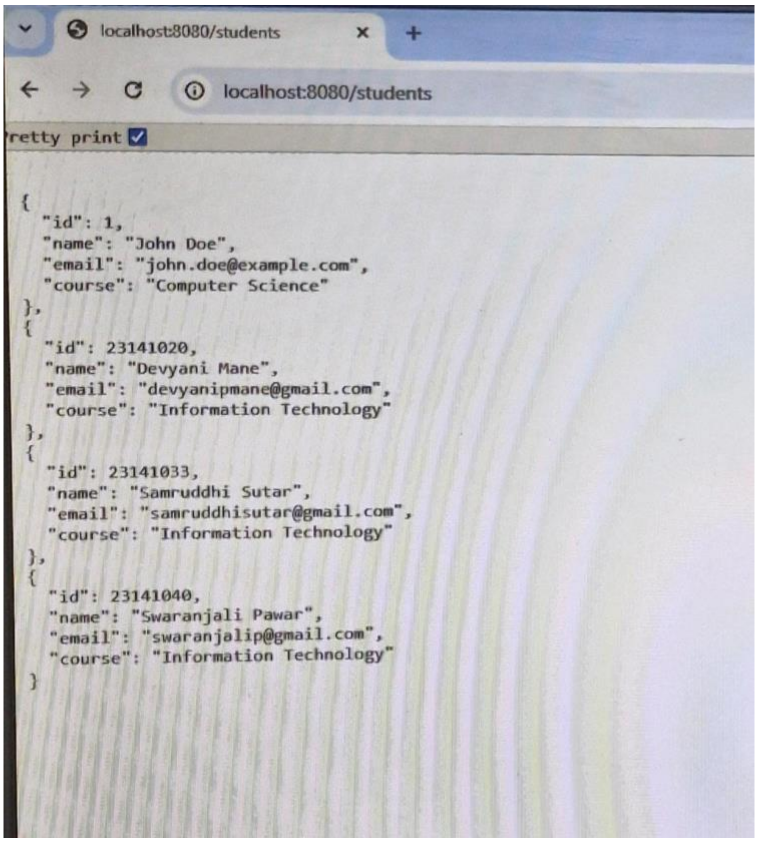
```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
	<modelVersion>4.0.0</modelVersion>
	<parent>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-parent</artifactId>
		<version>3.5.5</version>
		<relativePath/> <!-- lookup parent from repository -->
	</parent>
	<groupId>com.example.studentdata</groupId>
	<artifactId>StudentDataApp3</artifactId>
	<version>0.0.1-SNAPSHOT</version>
	<name>StudentDataApp3</name>
	<description>Demo project for Spring Boot</description>
	<url/>
	<licenses>
		<license/>
	</licenses>
	<developers>
		<developer/>
	</developers>
	<scm>
		<connection/>
		<developerConnection/>
		<tag/>
		<url/>
	</scm>
	<properties>
		<java.version>21</java.version>
	</properties>
	<dependencies>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-data-jpa</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>com.h2database</groupId>
			<artifactId>h2</artifactId>
			<scope>runtime</scope>
```

```
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
                <dependency>
        <groupId>javax.persistence</groupId>
        <artifactId>javax.persistence-api</artifactId>
        <version>2.2</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

Enter in Terminal -

```
curl -X POST http://localhost:8080/students \
-H "Content-Type: application/json" \
-d '{"id": 1,"name": "John Doe","email": "john.doe@example.com","course": "Computer Science"}'
```

Output:-



```
{
  "id": 1,
  "name": "John Doe",
  "email": "john.doe@example.com",
  "course": "Computer Science"
},
{
  "id": 23141020,
  "name": "Devyani Mane",
  "email": "devyanipmane@gmail.com",
  "course": "Information Technology"
},
{
  "id": 23141033,
  "name": "Samruddhi Sutar",
  "email": "samruddhisutar@gmail.com",
  "course": "Information Technology"
},
{
  "id": 23141040,
  "name": "Swaranjali Pawar",
  "email": "swaranjalip@gmail.com",
  "course": "Information Technology"
}
```

Conclusion

I have set up a simple Spring Boot application using **Eclipse** to manage student data with RESTful APIs. You can easily expand this by adding features like validation, pagination, authentication, and more.