

Introduction to Spring Boot

Overview

The Spring Framework is a widely used open-source Java framework that supports building enterprise-level applications. It is based on Inversion of Control (IoC) and Dependency Injection (DI) principles, which help to make the code more modular, reusable, and easier to maintain.

Spring Boot is a Spring Framework module that helps simplify the process of building Spring applications by providing pre-configured settings and a set of default dependencies. It eliminates the need for repetitive boilerplate code and allows developers to focus on the business logic of their applications.

Spring Boot has gained widespread adoption in the industry due to its ease of use, flexibility, and robustness. It is used by many well-known companies and organisations, including Netflix, Target, and Alibaba, to name a few.

Advantages of using Spring Boot

There are several advantages of using Spring Boot for building Java applications:

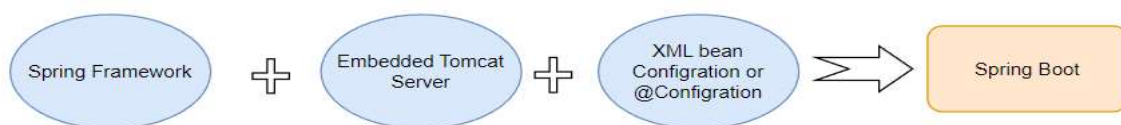
- **Faster Development:** Spring Boot provides a set of pre-configured dependencies and auto-configuration features that reduce the amount of boilerplate code needed for setting up a project. This lets developers focus on building business logic rather than worrying about infrastructure, resulting in faster development times.
- **Simplified Configuration:** Spring Boot provides a simple and intuitive way to configure the application, making it easier for developers to manage configuration files and settings.
- **Embedded Server:** Spring Boot includes an embedded web server, allowing developers to run and test their applications locally without deploying to a separate server. This speeds up the development process and reduces the time and effort required for deployment.
- **Modular Architecture:** Spring Boot provides a modular architecture allowing developers to choose the components they need for their projects. This provides greater flexibility and allows developers to create lightweight applications with only the necessary components.
- **Cloud Support:** Spring Boot provides out-of-the-box support for cloud deployment, making it easy to deploy applications to popular cloud platforms like AWS, Google Cloud, and Microsoft Azure.
- **Large Community:** Spring Boot has a large and active community of developers and users, which means there is a wealth of resources available, including documentation,

forums, and tutorials. This makes it easier for developers to find solutions to their problems and stay up-to-date with the latest best practices.

Spring vs Spring Boot

Spring and Spring Boot are popular frameworks for building Java applications but differ in their approach and functionality. Here are a few key differences between the two:

- **Configuration:** In Spring, you must manually configure everything from scratch, including the application server, dependencies, and other components. However, with Spring Boot, you can use pre-configured dependencies and auto-configuration features that save you time and effort.
- **Convention over Configuration:** Spring Boot follows a "Convention over Configuration" approach, providing default configurations and settings for various components. This approach reduces the need for manual configuration, thus making development faster and more efficient.
- **Ease of Use:** Spring Boot is designed to make application development faster and easier. It provides an embedded web server, auto-configuration, and other features that simplify development. Spring, on the other hand, requires more manual setup and configuration.
- **Dependency Management:** Spring Boot simplifies dependency management by providing a set of pre-configured dependencies that you can use in your application. Spring, on the other hand, requires you to manage dependencies manually.
- **Scope:** Spring is a comprehensive framework that covers a wide range of functionalities, including dependency injection, data access, web development, and more. Spring Boot, on the other hand, is a lightweight framework that focuses mainly on web development.



Overall, Spring Boot simplifies the development process by providing pre-configured dependencies and auto-configuration features. On the other hand, Spring offers more flexibility and control but requires more manual setup and configuration.

Introduction to Spring Boot

Overview

The Spring Framework is a widely used open-source Java framework that supports building enterprise-level applications. It is based on Inversion of Control (IoC) and Dependency Injection (DI) principles, which help to make the code more modular, reusable, and easier to maintain.

Spring Boot is a Spring Framework module that helps simplify the process of building Spring applications by providing pre-configured settings and default dependencies. It eliminates the need for repetitive boilerplate code and allows developers to focus on the business logic of their applications.

Spring Boot has gained widespread adoption in the industry due to its ease of use, flexibility, and robustness. It is used by many well-known companies and organisations, including Netflix, Target, and Alibaba, to name a few.

Advantages of using Spring Boot

There are several advantages of using Spring Boot for building Java applications:

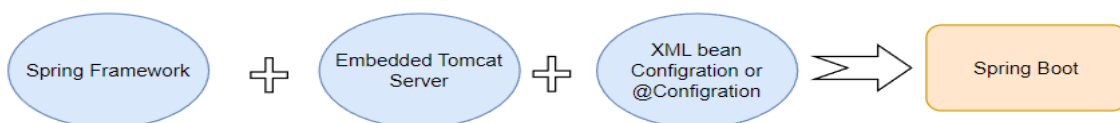
- **Faster Development:** Spring Boot provides a set of pre-configured dependencies and auto-configuration features that reduce the boilerplate code needed for setting up a project. This lets developers focus on building business logic rather than worrying about infrastructure, resulting in faster development times.
- **Simplified Configuration:** Spring Boot provides a simple and intuitive way to configure the application, making it easier for developers to manage configuration files and settings.
- **Embedded Server:** Spring Boot includes an embedded web server, allowing developers to run and test their applications locally without deploying to a separate server. This speeds up the development process and reduces the time and effort required for deployment.
- **Modular Architecture:** Spring Boot provides a modular architecture allowing developers to choose the components they need for their projects. This provides greater flexibility and will enable developers to create lightweight applications with only the necessary features.
- **Cloud Support:** Spring Boot provides out-of-the-box support for cloud deployment, making it easy to deploy applications to popular cloud platforms like AWS, Google Cloud, and Microsoft Azure.
- **Large Community:** Spring Boot has a large and active community of developers and users, which means there is a wealth of resources available, including documentation,

forums, and tutorials. This makes it easier for developers to find solutions to their problems and stay up-to-date with the latest best practices.

Spring vs Spring Boot

Spring and Spring Boot are popular frameworks for building Java applications but differ in their approach and functionality. Here are a few key differences between the two:

- **Configuration:** In Spring, you must manually configure everything from scratch, including the application server, dependencies, and other components. However, with Spring Boot, you can use pre-configured dependencies and auto-configuration features that save you time and effort.
- **Convention over Configuration:** Spring Boot follows a "Convention over Configuration" approach, providing default configurations and settings for various components. This approach reduces the need for manual configuration, thus making development faster and more efficient.
- **Ease of Use:** Spring Boot is designed to make application development faster and easier. It provides an embedded web server, auto-configuration, and other features that simplify development. Spring, on the other hand, requires more manual setup and configuration.
- **Dependency Management:** Spring Boot simplifies dependency management by providing a set of pre-configured dependencies that you can use in your application. Spring, on the other hand, requires you to manage dependencies manually.
- **Scope:** Spring is a comprehensive framework that covers a wide range of functionalities, including dependency injection, data access, web development, and more. Spring Boot, on the other hand, is a lightweight framework that focuses mainly on web development.



Overall, Spring Boot simplifies development by providing pre-configured dependencies and auto-configuration features. On the other hand, Spring offers more flexibility and control but requires more manual setup and configuration.

Inversion of Control

The foundation of the Spring Framework is the Spring IoC(Inversion of Control) container. The objects are created, their dependencies are configured and assembled, and their complete life cycle is managed.

It obtains the objects' details from an XML configuration file, Java code, Java annotations, or a Java POJO class. These things are referred to as Beans. Since developers are not managing Java objects' lifecycles, this phenomenon is known as an "inversion of control."

A. Using Interfaces

We start by creating interfaces of `Table`:

```
public interface Table {  
    String showDetails();  
}
```

We can define two implementations of this interface called ShortTable and LongTable:

```
public class ShortTable implements Table{  
    public void showDetails() {  
        return "the table has height " + this.height + " and length"+ this.length;  
    }  
}
```

```
public class LongTable implements Table{  
    public void showDetails() {  
        return "the table has length " + this.length + " height "+ this.height;  
    }  
}
```

And in the main class, we can create instances of either ShortTable or LongTable:

```
public class Main {  
    public static void main(String[] args) {  
        System.out.println("Welcome ! please use a size of table");  
        Scanner scanner = new Scanner(System.in);  
        String size = scanner.nextLine();  
        Table shortTable = new ShortTable();  
        Table longTable = new LongTable();  
        if(size.equals("long")) {  
            System.out.println(longTable.showDetails());  
        }  
        else {  
            System.out.println(shortTable.showDetails());  
        }  
    }  
}
```

B. Using ApplicationContext

This approach defines the objects and their dependencies in an XML file. For example, we can define the same ShortTable and LongTable classes. Then, we define beans in an XML file called applicationContext.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"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="shortTable" class="com.rahuMohan.testingSpringDemo.ShortTable">
    </bean>

    <bean id="longTable" class="com.rahuMohan.testingSpringDemo.LongTable">
    </bean>
</beans>
```

This XML file defines a bean for the ShortTable and a bean for LongTable.

In our Spring Boot application, we can use the ClassPathXmlApplicationContext to load the applicationContext.xml file and retrieve the MessageClient bean:

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Welcome ! please use a size of table");
        Scanner scanner = new Scanner(System.in);
        String size = scanner.nextLine();
        ClassPathXmlApplicationContext context =
            new
ClassPathXmlApplicationContext("applicationContext.xml");
        Table longTable = (Table) context.getBean("longTable");
        Table shortTable = (Table) context.getBean("shortTable");
        if(size.equals("long")) {
            System.out.println(longTable.showDetails());
        } else {
            System.out.println(shortTable.showDetails());
        }
    }
}
```

In this example, we load the applicationContext.xml file using the ClassPathXmlApplicationContext. We retrieve the shortTable bean using its id and call the showDetails() method.

Both approaches demonstrate how Spring Boot's IoC container can manage object creation leading to more flexible and maintainable code.

Port

In computer networking, a port is a communication endpoint or interface software applications use to send and receive data. Ports are identified by unique numbers called port numbers assigned to specific services or processes running on a computer or network device.

Port numbers range from 0 to 65535. They are divided into three ranges:

- **Well-known ports (0-1023):** These ports are reserved for specific services or protocols defined by the Internet Assigned Numbers Authority (IANA). For example, port 80 is commonly used for HTTP (Hypertext Transfer Protocol), port 443 for HTTPS (HTTP Secure), and port 25 for SMTP (Simple Mail Transfer Protocol).
- **Registered ports (1024-49151):** These ports are assigned by IANA for specific services or applications, but they are less widely recognised or standardised than the well-known ports. They are often used by client applications or server software for various purposes.
- **Dynamic or private ports (49152-65535):** These ports can be used by any application or service on an ad-hoc basis. They are typically allocated dynamically by the operating system when an application requests a port for communication.

On the other hand, **server port** refers to the specific port number that a server process is configured to listen on for incoming connections. When a client application wants to communicate with a server, it initiates a connection by specifying the server's IP address and the corresponding port number. The server, listening on that port, then accepts the incoming connection and establishes communication with the client.

By default, the spring boot application runs on port **8080**. However, you can configure it to run on any other port in the range of (1024 - 49151), i.e. in the category of registered ports, if any different server runs on 8080.

XML Configuration Tags

Here is a brief overview of the XML configuration tags and attributes commonly used in Spring Boot:

- **<beans>** tag defines a new bean in the Spring container. The id attribute is used to specify a unique identifier for the bean, and the class attribute is used to specify the fully qualified class name of the bean's implementation.
- **<context:component-scan>** tag enables component scanning in the Spring application context. The base-package attribute is used to specify the package where the Spring container should look for components to be registered as beans.

- **<property>** tag injects values into a bean's properties. The name attribute is used to specify the name of the property to be injected, and the value attribute is used to specify the value to be injected.
- **<constructor-arg>** tag injects values into a bean's constructor arguments. The value attribute is used to specify the value to be injected, and the index attribute is used to specify the index of the constructor argument.
- **<util:properties>** tag defines a set of properties as a bean. The location attribute is used to specify the location of the properties file, and the id attribute is used to specify the unique identifier of the bean.

These are just a few examples of the XML configuration tags and attributes used in Spring Boot. There are many more available depending on the specific needs of your application. However, it's worth noting that Spring Boot now encourages using Java-based configuration rather than XML configuration for greater flexibility and maintainability.

Conclusion

In conclusion, the Introduction to Spring Boot overviews the Spring framework, highlighting its key features and benefits for developers. It is designed to simplify the process of building and deploying Java-based web applications. It provides various tools and features that make it easier to manage dependencies, configure application settings, and test and deploy code.

In addition, the concept of Inversion of Control (IoC) using the ***ClassPathXmlApplicationContext*** is fundamental in the Spring framework. It lets developers decouple their code from specific implementation details, allowing them to write more flexible and modular applications. Using the ***ClassPathXmlApplicationContext***, developers can define and manage object dependencies within their applications, reducing the amount of boilerplate code and making it easier to maintain and scale applications over time.

Instructor Codes

- [Table Application](#)

References:

1. [Spring official Documentation](#)
2. [Port](#)
3. [XML](#)
4. [XML Boilerplate](#)
5. [Spring](#)
6. [Spring Vs Spring Boot](#)
7. [Advantages of Spring Boot over Spring](#)
8. [IOC](#)

Parameterised Beans

Introduction:

In Spring Framework, dependency injection is a core concept; constructor injection is one of the popular ways to inject dependency. If we want to inject a bean that takes specific parameters, we can do so in Spring Framework using the parameterised beans concept.

Both techniques have their benefits and drawbacks, and it is essential to understand the differences between them to choose the one that fits the specific requirements of an application. We will explore parameterised beans and constructor injection in detail and provide examples to illustrate their differences. We will also use a real-life example of an employee with two dependencies - salaries and a department - to demonstrate the practical application of these concepts. We will be using XML configuration to define beans and inject dependencies.

Constructor Injection:

Constructor injection is a popular technique used in Spring to inject dependencies into a class via its constructor. It provides a way to create and initialise objects with all the dependencies required to function correctly. Let us consider an example of an Employee class with two dependencies - Salary and Department - which are injected using constructor injection.

```
public class Employee {  
  
    private final Salary salary;  
    private final Department department;  
  
    public Employee(Salary salary, Department department) {  
        this.salary = salary;  
        this.department = department;  
    }  
  
    public void displayDetails() {  
        System.out.println("Salary: " + salary.getAmount());  
        System.out.println("Department: " + department.getName());  
    }  
}
```

In this example, the Employee class has two dependencies - Salary and Department - that are injected using constructor injection. The constructor takes two arguments, the Salary and Department objects, and assigns them to the respective instance variables. The displayDetails() method can then access these instance variables to display the employee's details.

Parameterised Beans:

Parameterised beans are a specific implementation of constructor injection that involves creating beans with a constructor that takes one or more parameters. These parameters are typically used to customise the bean's behaviour in some way, such as providing configuration information or injecting dependencies that are known in runtime. In the case of the Employee example, we can create parameterised beans for the Salary and Department classes to provide additional information to the Employee class.

Employee Example:

```
...  
public class Salary {  
  
    private final double amount;  
  
    public Salary(double amount) {  
        this.amount = amount;  
    }  
  
    public double getAmount() {  
        return amount;  
    }  
}  
...
```

```
...  
public class Department {  
  
    private final String name;  
  
    public Department(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
}
```

```
...
```

In these examples, we have created parameterised beans for the Salary and Department classes. Each bean has a constructor that takes a single parameter - salary amount and department name. The value of these parameters can be set at runtime to customise the behavior of the beans. For example, we can create a Salary bean with a value of 5000:

```
...  
<bean id="salary" class="com.example.Salary">  
    <constructor-arg value="5000.0"/>  
</bean>  
...
```

We can then inject this Salary bean into the Employee class using constructor injection, as follows:

```
...  
<bean id="employee" class="com.example.Employee">  
    <constructor-arg ref="salary"/>  
    <constructor-arg ref="department"/>  
</bean>  
...
```

In this example, the Employee bean is created with the Salary bean injected via constructor injection. The value of the Salary bean is set to 5000.0, and the Department bean is injected at runtime.

Let us take o more examples to understand the concept. Consider a blog application as an example. A typical blog application might have a class called BlogPost that represents a blog post. The BlogPost class might have two dependencies - a User object representing the author of the blog post and a Category object representing the blog post's category.

To inject these dependencies using constructor injection, we could define a constructor in the BlogPost class that takes two arguments - a User object and a Category object - like this:

Blog Example

```
// User Class  
public class User {  
    private String username;  
    private String email;  
  
    public User(String username, String email) {  
        this.username = username;  
        this.email = email;  
    }  
}
```

```

        // getters and setters
    }

    // Category Class
    public class Category {
        private String name;

        public Category(String name) {
            this.name = name;
        }

        // getters and setters
    }

    // Blog Post Class
    public class BlogPost {
        private User author;
        private Category category;

        public BlogPost(User author, Category category) {
            this.author = author;
            this.category = category;
        }

        // getters and setters
    }

```

This example defines the User and Category classes with constructors that take the necessary arguments. We've also updated the BlogPost class to use the User and Category objects passed in via the constructor.

To use this constructor, we would define the User and Category beans first and then define the BlogPost bean, passing in the User and Category beans as constructor arguments like this:

```

<bean id="author" class="com.example.User">
    <constructor-arg value="johndoe"/>
    <constructor-arg value="johndoe@example.com"/>
</bean>

<bean id="category" class="com.example.Category">
    <constructor-arg value="Technology"/>
</bean>

<bean id="blogPost" class="com.example.BlogPost">
    <constructor-arg ref="author"/>
    <constructor-arg ref="category"/>

```

```
</bean>
```

In this example, we've defined a BlogPost bean with two constructor arguments - a User object and a Category object - and we've defined the values of these arguments directly in the XML configuration file. By using constructor injection or parameterised beans, we can easily inject dependencies into our classes and configure our application with ease.

Comparison:

Now that we have seen parameterised beans and constructor injection examples let us compare the two techniques based on their benefits and drawbacks.

1. **Flexibility:** Constructor injection is more flexible than parameterised beans because it can inject any type of dependency. On the other hand, Parameterised beans can only inject dependencies that can be instantiated with a constructor that takes parameters. However, parameterised beans offer more flexibility than regular beans as you can customise their behavior at runtime by passing different values for constructor parameters.
2. **Ease of use:** Constructor injection is generally easier than parameterised beans. You don't need to define beans with constructor arguments explicitly. You can specify the beans in the XML configuration file and let Spring handle the injection automatically. Parameterised beans require more work as you must define the beans explicitly and provide constructor arguments while creating the beans.
3. **Testability:** Constructor injection is more testable than parameterised beans because creating mock objects for dependencies is easier. Parameterised beans require creating a new bean for each test case, which can be cumbersome and time-consuming.
4. **Configuration management:** Parameterized beans provide better configuration management because you can specify the values of constructor parameters in the XML configuration file. This makes it easier to manage configuration changes and ensures that the same configuration is used across different bean instances.

Conclusion:

In conclusion, parameterised beans and constructor injection are valuable techniques for injecting dependencies in Spring Framework. Constructor injection is more flexible and easier to use, whereas parameterised beans provide more customisation and better configuration management.

When choosing between the two techniques, it is essential to consider the application's specific requirements and choose the best method. Using the real-life example of an *Employee class with dependencies on Salary and Department classes*, we have illustrated the practical application of these concepts and demonstrated how they could be implemented using XML configuration in Spring.

Dependency Injection

Overview

Dependency injection is a design pattern that separates an object's creation and usage by providing its dependencies externally rather than having the object create them internally.

No matter how simple, every application consists of multiple objects that work together to provide a unified user experience. We use the dependency injection principle to get the objects to collaborate and achieve a common objective.

What Is Dependency Injection

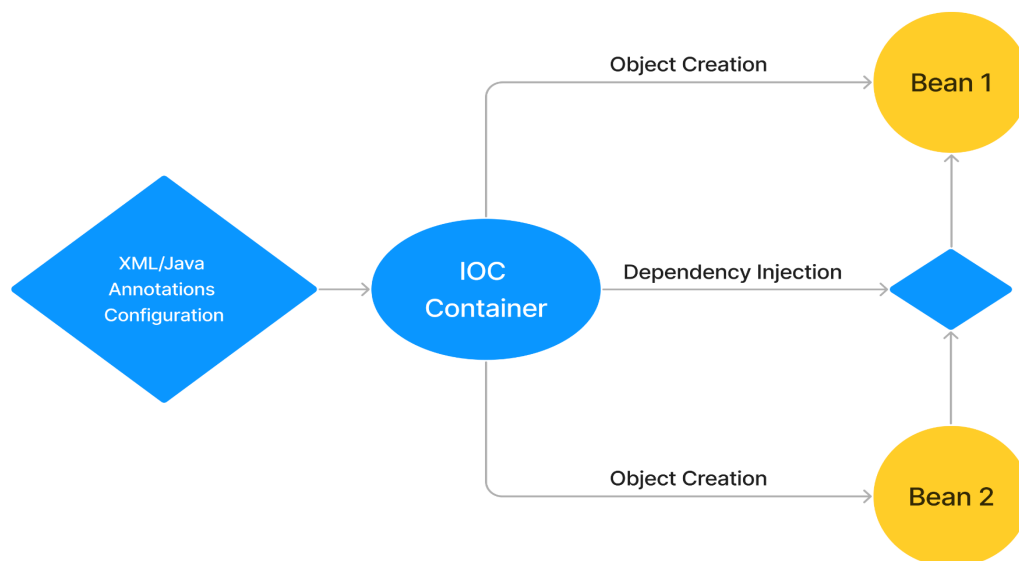
- Dependency injection (DI) is a technique objects use to specify their dependencies, such as other objects they interact with. This is achieved by providing constructor arguments, properties, or arguments to a factory method.
- The container then provides the necessary dependencies when creating the object, which is the opposite of traditional object instantiation, where the object determines its dependencies' location or creation, often using the Service Locator pattern. This approach is commonly known as Inversion of Control.
- Dependency injection achieves loose coupling between objects by removing the responsibility of object instantiation and dependency management from the object itself. Instead, the dependencies are provided externally, typically by a DI container or framework, allowing the object to focus solely on its functionality.
- This means that objects can be easily replaced or updated without affecting other system parts, as long as they adhere to the same interface or contract. By reducing the interdependence between objects, DI enables greater flexibility and maintainability of the system, making it easier to test, extend, and evolve.

How Dependency injection works

The following is an overview of how DI works:

1. **Object Definition:** First, a developer defines the objects and their dependencies using a DI framework or container. The dependencies are constructor arguments, properties, or factory methods.
2. **DI Container Configuration:** The DI container is then configured to manage the dependencies of the objects. This involves registering the objects and their dependencies with the container.
3. **Object Creation:** When an object is requested directly or as a dependency of another object, the container creates it and provides its dependencies.
4. **Dependency Injection:** The container injects the dependencies into the object's constructor or properties, allowing the object to access and use them.

By providing dependencies externally, DI reduces the coupling between objects and improves the modularity and maintainability of the code. It also makes testing easier, as dependencies can be easily mocked or substituted with test-specific implementations.



Types of Dependency Injection

A. Constructor-based Dependency Injection

Constructor-based dependency injection is a dependency injection in which a class's dependencies are provided through its constructor.

In this approach, the class's constructor takes one or more arguments that represent the required dependencies, which are passed in when an instance of the class is created. The constructor then stores these dependencies as instance variables that can be used throughout the class.

Car Dealership Example:

We start by creating a Car interface:

```
public interface Car{
    String showDetails();
}
```

We can define two implementations of this interface called FamilyCar and SportsCar

```
public class FamilyCar implements Car{

    private String owner;
    private String brand;

    public FamilyCar(String owner, String brand){
        this.owner = owner;
        this.brand = brand;
    }

    public String showDetails() {
        return this.owner + " has a family car of brand " + this.brand;
    }

}
```

```
public class SportsCar implements Car {

    private String owner;
    private String brand;

    public SportsCar(String owner, String brand){
        this.owner = owner;
        this.brand = brand;
    }

    public String showDetails() {
        return this.owner + " has a Sports car of brand " + this.brand;
    }
}
```

We Define objects and their dependencies in XML

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="sports" class="com.example.demo.SportsCar">
        <constructor-arg name="owner" value="Rahul"/>
        <constructor-arg name="brand" value="Tata Neo"/>
    </bean>
    <bean id="family" class="com.example.demo.FamilyCar">
        <constructor-arg name="owner" value="Rahul"/>
        <constructor-arg name="brand" value="Audi A8"/>
    </bean>
</beans>
```

In the main method, we take the user's name and car choice and show details accordingly.

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hi welcome, which type of car do you want");
        Scanner scanner = new Scanner(System.in);
        String type = scanner.nextLine();
        ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
        Car car = (Car) context.getBean(type);
        System.out.println(car.showDetails());
    }
}
```

B. Setter-based Dependency Injection

Setter injection is a dependency injection where dependencies are injected into an object using setter methods.

With setter injection, an object is first created using a no-argument constructor, and then its dependencies are set using setter methods. Setter injection can be helpful when we have optional dependencies or when we need to inject multiple dependencies into an object.

In some cases, it can also be easier to read and maintain, as it separates the dependency injection from the object's construction. However, it can be less explicit than constructor injection, making it harder to ensure that all dependencies are correctly set.

In continuation to the above example, we convert it into a setter injection

```
public class FamilyCar implements Car{

    private String owner;
    private String brand;

    public void setOwner(String owner){
        this.owner = owner;
    }
    public void setBrand(String brand){
        this.brand = brand;
    }

    public String showDetails() {
        return this.owner + " has a family car of brand " + this.brand;
    }
}
```

```
public class SportsCar implements Car {

    private String owner;
    private String brand;

    public void setOwner(String owner){
        this.owner = owner;
    }
    public void setBrand(String brand){
        this.brand = brand;
    }

    public String showDetails() {
        return this.owner + " has a Sports car of brand " + this.brand;
    }
}
```

```
}
```

We Define objects and their dependencies in XML using the property tag

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="sports" class="com.example.demo.SportsCar">
        <property name="owner" value="Rahul"/>
        <property name="brand" value="BMW V8"/>
    </bean>
    <bean id="family" class="com.example.demo.FamilyCar">
        <property name="owner" value="Rahul"/>
        <property name="brand" value="Ertiga"/>
    </bean>
</beans>
```

In the main method, we take the user's name and car choice and show details accordingly.

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hi welcome, which type of car do you want");
        Scanner scanner = new Scanner(System.in);
        String type = scanner.nextLine();

        ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
        Car car = (Car) context.getBean(type);
        System.out.println(car.showDetails());
    }
}
```

C. Field Injection

Field injection is a dependency injection technique in Spring Boot that involves injecting dependencies directly into class fields. To use field injection, annotate the field with the `@Autowired` annotation.

When the application starts, Spring Boot will automatically scan for all `@Autowired` fields and inject the appropriate dependencies. Field injection is a convenient way to inject dependencies, but it can make testing more difficult since dependencies cannot be easily swapped out for mocks.

It is generally recommended to use a constructor or setter injection instead, as it they more explicit and testable. However, in some cases, such as with third-party libraries or legacy code, field injection may be necessary.

Example:

```
public class Car{  
  
    @Autowired  
    private Engine engine;  
  
}
```

Common Dependency Errors

- **Circular dependencies:** Occurs when two or more classes depend on each other. It can result in a deadlock where none of the classes can be instantiated. To avoid this, designing classes with clear responsibilities and avoiding circular dependencies is crucial.
- **Missing dependencies:** This occurs when a required dependency is not available. This can happen when the dependency is not configured correctly or is not present in the application context. It is crucial to ensure that all dependencies are perfectly defined and available.
- **Ambiguous dependencies:** This occurs when multiple dependencies match the required dependency type. This can happen when various interface or abstract class implementations exist. Qualifying dependencies must be qualified using annotations or configured correctly to avoid this.
- **Incorrect dependencies:** Occurs when the wrong dependency is injected into a class. This can happen when the dependency type must be correctly matched to the required type. Ensuring that the correct dependencies are injected in essential classes is essential.
- **Inconsistent dependencies:** This occurs when the dependencies injected into a class are inconsistent with the dependencies used by the class. This can happen when the dependencies need to be appropriately synchronised or when different versions of dependencies are used. To avoid this, it is essential to use a consistent set of dependencies throughout the application.

Instructor Codes

- [Car Dealership Application](#)

References:

1. [Official spring documentation.](#)
2. [Constructor Injection](#)
3. [Setter Injection](#)
4. [Common Exceptions in Dependency Injection](#)

@PostConstruct and @PreDestroy Annotations

Introduction

In Spring, the usage of the *@PostConstruct* and *@PreDestroy* annotations is indeed correct. However, the information regarding the destructor is not accurate. In Java, the concept of destructors, also known as finalizers, is different from the way Spring manages the lifecycle of beans.

Here's the corrected information regarding the destructor in Java and the differences between constructors, *@PostConstruct*, *@PreDestroy*, and destructors:

@PostConstruct

@PostConstruct is an annotation provided by the Java EE and Spring frameworks. When you annotate a method with *@PostConstruct*, it is executed after the bean has been initialized by the container. It is often used for performing any initialization tasks that require the fully constructed bean instance. For example, you can use it to initialize database connections or establish network connections.

@PreDestroy

@PreDestroy is another annotation provided by Java EE and Spring. When you annotate a method with *@PreDestroy*, it is executed just before the bean is destroyed by the container. It is often used for performing any cleanup tasks before the bean is removed from the container. For example, you can use it to close open resources like database connections or release acquired locks.

Destructor

In Java, a destructor, also known as a finalizer, is a special method defined in a class that is automatically called by the garbage collector when the object is being garbage-collected. The purpose of the finalizer is to perform any necessary cleanup or resource release before the object is destroyed. However, using destructors/finalizers is generally discouraged in Java due to their unpredictable execution timing and potential performance issues.

In the context of Spring, the framework does not directly provide support for defining and executing destructors/finalizers for beans. Instead, Spring encourages the use of *@PreDestroy* annotation or implementing the *DisposableBean* interface to handle cleanup tasks before the bean is destroyed.

To Summarise:

- Constructor: Used for object instantiation and dependency injection.
- *@PostConstruct*: Used for initialization tasks after object construction.
- *@PreDestroy*: Used for cleanup tasks before object destruction.
- Destructor (Finalizer): Not commonly used in Spring and discouraged in Java due to their unpredictable execution. Spring provides alternative mechanisms like *@PreDestroy* and *DisposableBean* for cleanup tasks.

Bean and Bean Lifecycle

What are Beans?

Spring Boot beans are objects managed by the Spring framework's Inversion of Control (IoC) container. These beans can be used to define the various components of an application, such as services, repositories, controllers, etc.

Here are some critical points about beans in Spring Boot:

- Beans are created by the Spring IoC container and managed throughout the application's lifecycle.
- Beans can be defined in several ways, including through Java configuration, XML configuration, and annotations.
- Beans can be wired together through dependency injection, allowing them to collaborate and work together within the application.
- Spring Boot provides several built-in beans, including the application context, the core container for managing beans, and the environment, which allows access to application properties and configuration.
- Beans can also be scoped, meaning specific context or scope determines their lifecycle. Spring Boot supports several scopes, including singleton, prototype, request, session, and WebSocket.
- Overall, beans are a fundamental concept in Spring Boot and are essential for building and managing complex applications in a scalable and maintainable way.

Bean Scope

Bean scope refers to the lifecycle and visibility of a bean within the Spring container. Spring Boot supports several bean scopes, including Singleton, Prototype, Request, Session, and WebSocket.

A. Singleton

Singleton is the default bean scope in Spring Boot. A singleton bean is created only once by the Spring IoC container, and a single instance of the bean is shared throughout the application. This means the container returns the same instance whenever the application requests the same bean.

The singleton bean scope is suitable for beans that are stateless and do not have any mutable state. Examples of such beans are utility or service classes that don't maintain any state. Singleton beans are naturally thread-safe, as they are only created once and shared among multiple threads. This can improve performance and reduce memory usage, as only one instance of the bean is created and maintained throughout the application's lifecycle.

For Example, **Social Media Application** that we saw in the lecture

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="simplePostList" class="SimplePostList" scope="singleton">
    </bean>

    <bean id="SimpleUser" class="simpleUser" scope="singleton">
    <property name="postList" ref="simplePostList" />
    </bean>

</beans>
```

B. Prototype

The prototype bean scope in Spring Boot creates a new instance of a bean every time the bean is requested from the container. This means that every time the application requests a prototype bean, the container creates a new instance of the bean instead of returning an existing instance.

The prototype bean scope is suitable for beans that maintain a state or have mutable properties and where multiple instances of the bean are required. Examples of such beans are command objects or form objects that must maintain state across multiple requests.

However, prototype beans can be memory-intensive and lead to performance issues if not appropriately managed, as a new instance of the bean is created every time it is requested. Also, prototype beans are not naturally thread-safe, so the application developer is responsible for ensuring thread safety if multiple threads access the same bean instance.

For Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="simplePost" class="SimplePost" scope="prototype">
    </bean>

    <bean id="simplePostList" class="SimplePostList" scope="singleton">
    </bean>

    <bean id="SimpleUser" class="simpleUser" scope="singleton">
    <property name="postList" ref="simplePostList" />
    </bean>

</beans>
```

C. Web Aware Scopes

C. 1 Request

The request bean scope in Spring Boot creates a new instance of a bean for every HTTP request made to the application. This means that whenever a request is made to the application, the container creates a new instance of the request-scoped bean, and the bean instance is available only within that specific request.

The request bean scope is suitable for beans that maintain a state-specific HTTP request, such as form data, user session information, or request-specific data that needs to be processed by multiple beans. This helps keep the application's state separate across numerous requests, improving the application's stability and reliability.

C. 2 Session

Session scope in Spring Boot is used to create a single instance of a bean for each user session. This means that each user accessing the application will have their unique instance of the bean, which will be created and managed by the Spring container.

When a bean is defined with session scope, a new instance is created for each HTTP session. This is useful when you want to maintain state information across multiple requests from the same user. For example, if your application has a shopping cart feature, you would like to retain the cart's contents for each user session. Defining the shopping cart bean with session scope would ensure each user has a unique cart instance.

D. Other Scopes

Some custom scopes are available in the Spring framework but not built-in in Spring Boot. The "Global Session" and "Application" are lesser-used custom scopes.

The **Java Servlet** specification (*which will study in brief in the upcoming lectures or refer to links in the later section*) provides the "**Global Session**" scope, and it defines a single bean definition of the lifecycle of a global HTTP session. This scope is valid only when used in a Servlet-based web application, such as a portlet context. When an application is built of portlets, each portlet has its session. Still, if you want to store variables globally for all portlets in your application, you can use the "Global Session" scope to create a single bean definition shared across all portlets.

The "**Application**" scope is not standard in Spring Boot or the Java Servlet specification. However, some third-party libraries or frameworks may define their custom scopes with different lifecycles and semantics, and one of them may be called the "Application" scope. When Spring creates a bean instance per web application runtime, it is similar to the Singleton scope, with one significant difference.

A Singleton scoped bean is a singleton per `ApplicationContext`, whereas an Application scoped bean is a singleton per `ServletContext`. It's important to note that multiple contexts exist for a single application.

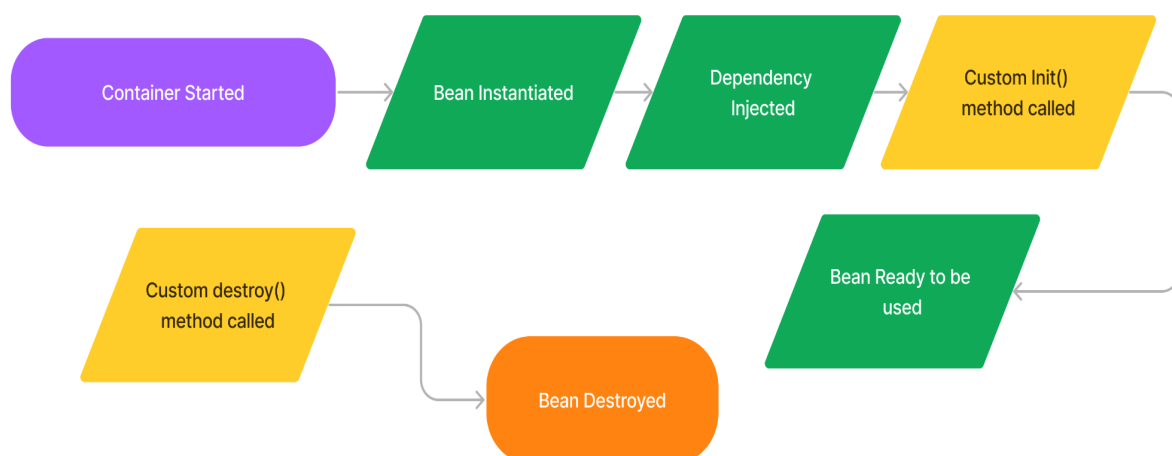
Bean Lifecycle

A bean's lifecycle in Spring Boot refers to the various phases, from creation to destruction. The bean lifecycle can be divided into the following phases:

- **Instantiation:** During this phase, a bean is created by invoking its constructor or a factory method.
- **Dependency Injection:** After instantiating a bean, any dependencies are injected using a constructor or setter injection.
- **Initialization:** Once all the dependencies have been injected, any initialisation logic that needs to be performed on the bean can be executed. This may include setting default values or configuring the bean. Developers can specify initialisation methods for a bean by using the `@PostConstruct` annotation on a method in the bean class method or by specifying an `init-method` attribute in the XML configuration file.
- **Usage:** During this phase, the bean performs its intended functionality.
- **Destruction:** When the bean is no longer needed, it is destroyed. This may involve releasing the bean's resources, such as file handles or network connections. Developers can specify destruction methods for a bean by using the `@PreDestroy` annotation on a method in the bean class method or by specifying a `destroy-method` attribute in the XML configuration file.

In Spring, the container responsible for managing the bean lifecycle is `ApplicationContext`. When the `ApplicationContext` is started, it creates and initializes all the beans defined in the application context, and when the application is shut down, it destroys all the beans.

Developers can customize the bean lifecycle by providing callback methods that Spring will invoke at specific phases in the bean lifecycle. These methods can be defined using annotations or in the XML configuration file.



For Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="simplePost" class="SimplePost" scope="prototype"
        destroy-method="destroy">
    </bean>

    <bean id="simplePostList" class="SimplePostList" scope="singleton"
        init-method="init" destroy-method="destroy">
    </bean>

    <bean id="SimpleUser" class="simpleUser" scope="singleton">
    <property name="postList" ref="simplePostList" />
    </bean>

</beans>
```

Instructor Codes

- [Social Media Application](#)

References:

1. [Official spring documentation.](#)
2. [Bean Scopes](#)
3. [Bean Lifecycle](#)
4. [Medium Article on bean lifecycle](#)
5. [Java Servlets](#)
6. [Session Scope](#)
7. [HTTP Request](#)
8. [Web Socket](#)
9. [Thread safety](#)

Spring and Annotations

Introduction

Annotations in Spring are metadata markers that provide additional information or instructions to the Spring framework. They are applied to classes, methods, fields, or method parameters and help configure various aspects of the application's behaviour, such as dependency injection, aspect-oriented programming, transaction management, and more.

Annotations in Spring make it possible to declaratively define the application's configuration and behaviour, reducing the need for explicit XML configuration files. Let's explore the different annotations in Spring:

A. @Component Annotation

The @Component annotation is a feature provided by several Java frameworks, such as Spring Framework, that allows you to declare a class as a component or bean. This annotation marks a class as a candidate for auto-detection and auto-configuration when the application context is created.

When annotating a class with @Component, it becomes a Spring bean registered in the application context. The Spring container then manages the lifecycle and dependencies of these beans.

Here are some key points about the @Component annotation:

- **Auto-detection:** When using component scanning in Spring, the framework automatically detects classes annotated with @Component and registers them as beans in the application context.
- **Dependency injection:** Components annotated with @Component can be injected into other classes using dependency injection mechanisms provided by the framework, such as @Autowired.
- **Customization:** The @Component annotation is a generic annotation used as a base annotation for more specific stereotypes like @Service, @Repository, or @Controller. These specialised annotations help semantically categorise the components and provide additional functionality or behaviour.
- **Configuration:** By default, Spring treats classes annotated with @Component as singleton beans. However, you can customise the bean's scope by using other annotations like @Scope or @RequestScope to change the instantiation and lifecycle behaviour.

- **Component scanning:** To enable component scanning and make Spring automatically detect and register the `@Component` annotated classes, you need to configure it in the Spring configuration file or use the appropriate annotations, such as `@ComponentScan`, on a configuration class.
- **XML-based configuration:** If you use XML-based configuration instead of annotations, you can still define a bean using the `<bean>` element and specify the class as a component using the `class` attribute.

For Example,

```
@Component("javaInstructor")
public class JavaInstructor implements Instructor {

    String name;
    String age;

    @Override
    public void setInstructorDetails(String name, String age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public String takeClass() {
        return "Hi my name is "+this.name+" and I will be your
java" + "instructor";
    }
}
```

In the example above, The `@Component` annotation is used to mark the `JavaInstructor` class as a Spring component. The optional parameter `"javaInstructor"` specifies the name or identifier for the bean in the Spring container. If no name is specified by default, the bean name is derived from the class name with the initial letter converted to lowercase.

Overall, the `@Component` annotation provides a convenient way to mark classes as Spring components or beans, enabling automatic dependency injection and lifecycle management by the Spring container.

B. @Autowire Annotation

The `@Autowired` annotation is a feature of the Spring Framework in Java that allows automatic dependency injection. When you annotate a field, constructor, or setter method with `@Autowired`, Spring will automatically resolve and inject the appropriate dependency at runtime.

Here are some key points about the `@Autowire` annotation:

- **Dependency injection:** You can annotate a field directly with `@Autowired` to indicate that Spring should inject a dependency into that field.
- **Dependency resolution:** When you use `@Autowired`, Spring will analyse the type of dependency and search for a matching bean in the application context. It will be injected if there's exactly one bean of that type. If multiple beans are of the same type, you can use additional annotations like `@Qualifier` to specify the bean to be injected.
- **Optional dependencies:** By default, `@Autowired` requires a dependency in the application context. However, you can make a dependency optional by using the required attribute of `@Autowired` and setting it to false. The dependency will be set to null if a matching bean is not found.
- **Qualifiers:** In scenarios with multiple beans of the same type, you can use the `@Qualifier` annotation in conjunction with `@Autowired` to specify the exact bean to be injected. The `@Qualifier` annotation can be applied to a field, constructor parameter, or setter method parameter.
- **Primary bean:** If multiple beans of the same type are available and you want to specify a primary bean that should be injected by default, you can use the `@Primary` annotation on that bean. When using `@Autowired`, the primary bean will be selected if no specific qualifier or name is provided.
- Here is an example of DI using `@Autowired` annotation,

```
@Component
public class PaidCourse implements Course{

    String courseName;

    @Autowired
    UserList userList;

    @Override
    public void setCourseDetails(String courseName) {
        this.courseName = courseName;
    }
}
```

```
    }

    @Override
    public UserList getUserList() {
        return this.userList;
    }
}
```

Overall, the `@Autowired` annotation simplifies the process of injecting dependencies by allowing Spring to wire beans together automatically. It promotes loose coupling and improves the maintainability and flexibility of your code.

C. `@Qualifier` Annotation

The `@Qualifier` annotation is used with the `@Autowired` annotation in the Spring Framework to specify the exact bean to inject when multiple beans of the same type are available.

Here are some key points about the `@Qualifier` annotation:

- **Multiple bean candidates:** In some cases, there may be multiple beans of the same type may be available in the application context. For example, if you have multiple implementations of an interface, Spring may not be able to determine which bean to inject automatically.
- **Qualifying a specific bean:** Using the `@Qualifier` annotation, you can specify the name or identifier of the bean that will be injected. The annotation allows you to disambiguate between multiple beans and provide clarity to Spring about which bean should be used.
- **Bean qualifier values:** The `@Qualifier` annotation can be applied to a field, constructor, or setter method parameter. You can provide a qualifier value as an argument to the `@Qualifier` annotation to indicate the specific bean to be injected. For example,

```
@Component
public class PaidCourse implements Course{

    String courseName;

    @Autowired
    @Qualifier("javaInstructor")
    Instructor courseInstructor;

    @Autowired
```

```
UserList userList;  
  
    // additional methods  
}
```

In the example above, the `@Qualifier("javaInstructor")` annotation is used to specify the bean with the name `"javaInstructor"` to be injected. It helps resolve any ambiguity when multiple beans of the `Instructor` type are available.

In the example below, an interface `Salary` with two implementations `FixedSalary` and `VariableSalary`, and `Salary`, is injected in the `Employee` class. We only have to inject `FixedSalary`. We use `@Qualifier("fixedSalary")`, i.e. qualifier annotation with the bean name to inject in string format.

```
// Salary interface  
public interface Salary {  
    void getSalary();  
}  
  
// FixedSalary class  
public class FixedSalary implements Salary {  
    //...  
}  
  
// VariableSalary class  
public class VariableSalary implements Salary {  
    //...  
}  
  
// Employee class  
public class Employee {  
    private String name;  
  
    @Autowired  
    @Qualifier("fixedSalary")  
    Salary salary;  
}
```

The `@Qualifier` annotation is a powerful tool for fine-grained control over dependency injection in Spring when dealing with multiple beans of the same type. It lets you specify the bean to inject, promoting clarity and resolving ambiguities in the dependency resolution process.

D. Dependency injection in the Main class,

The main method in the main class of a Spring Boot application, is static. Therefore, that dependency cannot be used in the main method. To use that dependency, we use the CommandLineRunner interface. The CommandLineRunner interface is commonly used in Spring Boot to execute code after the application context is initialised and the main method is invoked.

After the application context is initialised and the main method is invoked, the run method of CommandLineRunner will be automatically executed by Spring Boot. This provides an opportunity to perform additional actions or invoke methods that depend on the injected dependencies.

Let's take an example of a vaccine application where we want to inject a User dependency in our application using annotation. Here we are going to inject the User directly into our application rather than fetching from context.

```
@SpringBootApplication
public class VaccineWithAnnotationApplication implements CommandLineRunner {

    @Autowired
    @Qualifier("Father")
    User user;

    public static void main(String[] args) {
        // application logic
    }
    @Override
    public void run(String... args) throws Exception {
        System.out.println("Please choose your vaccine type:");
        System.out.println("1. Covaxin\n2. Covishield\n3.Moderna");
        int vaccineChoice = scanner.nextInt();
        scanner.nextLine();
        String myVaccine;
        switch (vaccineChoice) {
            case 1 -> myVaccine = "Covaxin";
            case 2 -> myVaccine = "Covishield";
            case 3 -> myVaccine = "Moderna";
            default -> {
                System.out.println("Invalid choice.");
                return;
            }
        }

        this.user.setVaccine(myVaccine);
    }
}
```

This is useful when we want to insert specific data in the database when the application is ready, integrate some external system or schedule some tasks at application startup etc.

E. @Scope Annotation

The `@Scope` annotation is used in the Spring Framework to specify the scope or lifecycle of a bean. It allows you to define how bean instances should be created, managed, and destroyed within the Spring container.

Here are some key points about the `@Scope` annotation:

- **Singleton scope (default):** By default, Spring beans have a singleton scope, meaning that only one instance of the bean is created and shared across the entire application context. The `@Scope` annotation can be used to explicitly define the singleton scope, although it is not necessary as it is the default behaviour. For example:
- **Prototype scope:** The `@Scope` annotation can specify a bean's prototype scope. In this case, a new instance of the bean is created whenever it is requested from the container. Each instance is independent of others and has its state. For example,

```
@Component
@Scope("prototype")
public class SimpleUser implements User {

    String name;
    String age;
    String location;
    String collegeName;

    @Override
    public String getUserDetails() {
        return this.name + " age:" + this.age;
    }

    @Override
    public void setUserDetails(String name, String age, String
location, String collegeName) {
        this.name = name;
        this.age = age;
        this.collegeName = collegeName;
        this.location = location;
    }
}
```

In the example above, `@Scope("prototype")` is used to specify the scope of bean of class `SimpleUser` should be a prototype.

The `@Scope` annotation allows you to control the lifecycle and availability of beans within the Spring container. By specifying different scopes, you can define whether beans should be singletons, prototypes, or tied to specific contexts like requests or sessions. It provides flexibility in managing the lifecycle and state of beans based on your application's requirements.

F. `@PostConstruct` Annotation

The `@PostConstruct` annotation is used in the Java EE and Spring frameworks to annotate a method that needs to be executed after dependency injection and bean initialisation. It is often a callback method for a bean's initialisation tasks.

Here's a description of the `@PostConstruct` annotation and its common uses:

- **Initialization callback:** By annotating a method with `@PostConstruct`, you indicate to the framework that this method should be invoked after the bean has been constructed, its dependencies have been injected, and before it is put into service.
- **Method requirements:** The method annotated with `@PostConstruct` should not have any arguments and must be non-private. It can have any return type, although void is commonly used since the method is typically used for initialisation rather than returning a value.
- **Order of execution:** When a bean is instantiated and all its dependencies are injected, the method annotated with `@PostConstruct` is automatically called by the container or framework. This ensures that the initialisation tasks defined in the method are executed at the appropriate time.
- **Initialization tasks:** The method annotated with `@PostConstruct` can be used to perform various initialisation tasks, such as setting up resources, establishing connections, initialising variables, or any other operations required to prepare the bean for use.
- **Multiple methods:** Annotating multiple methods with `@PostConstruct` within a single bean is possible. In such cases, the order of execution is not guaranteed. If you need specific ordering, you can use the `@Order` annotation or implement the `org.springframework.core.Ordered` interface.

For example,

```
@Component("javaInstructor")
public class JavaInstructor implements Instructor {

    String name;
    String age;

    @PostConstruct
    public void init() {
```

```
        System.out.println("Java instructor bean created");  
    }  
}
```

The `@PostConstruct` annotation provides a convenient way to define initialisation behaviour for beans in Java EE and Spring applications. It allows you to centralise and manage the initialisation tasks of the bean, ensuring they are executed appropriately in the bean's lifecycle.

G. `@PreDestroy` Annotation

The `@PreDestroy` annotation is used in Java EE and Spring applications to annotate a method that should be called before a managed bean is destroyed or removed from service. It is commonly used as a callback method to perform cleanup or release resources held by a bean.

Here's a description of the `@PreDestroy` annotation and its common uses:

- **Destruction callback:** By annotating a method with `@PreDestroy`, you indicate to the container or framework that this method should be invoked before the bean is destroyed or removed from service. It lets you perform necessary cleanup tasks or release the bean's resources.
- **Method requirements:** The method annotated with `@PreDestroy` should not have any arguments and must be non-private. It can have any return type, although void is commonly used since the method is typically used for cleanup rather than returning a value.
- **Order of execution:** When a managed bean is being destroyed, the method annotated with `@PreDestroy` is automatically called by the container or framework. This ensures that the cleanup tasks defined in the method are executed before the bean is removed from the service.
- **Resource cleanup:** The `@PreDestroy` method is often used to release resources such as open files, database connections, network connections, or other system resources held by the bean. It allows you to gracefully shut down or clean up resources before the bean is destroyed.
- **Bean lifecycle management:** The `@PreDestroy` annotation is part of the broader bean lifecycle management mechanism provided by Java EE and Spring. It complements the `@PostConstruct` annotation, which is used for initialisation tasks. Together, these annotations provide a way to perform actions at specific points in the lifecycle of a managed bean.

For example,

```
@Component("javaInstructor")
public class JavaInstructor implements Instructor {

    String name;
    String age;

    @PreDestroy
    public void cleanup() {
        System.out.println("Java instructor bean about to be
destroyed");
    }
}
```

The `@PreDestroy` annotation provides a convenient way to define cleanup behaviour for managed beans in Java EE and Spring applications. It allows you to centralise and manage resource cleanup for the bean, ensuring it is performed before it is destroyed or removed from service.

H. @SpringBootApplication Annotation

The `@SpringBootApplication` annotation is a convenience annotation provided by the Spring Boot framework. It is used to annotate the main class of a Spring Boot application to enable various auto-configuration and component scanning features.

Here are some key points about the `@SpringBootApplication` annotation:

- **Combination of annotations:** The `@SpringBootApplication` annotation combines three commonly used annotations in Spring Boot applications: `@Configuration`, `@EnableAutoConfiguration`, and `@ComponentScan`. By using `@SpringBootApplication`, you can apply these three annotations with a single annotation.
- **@Configuration:** The `@Configuration` annotation indicates that the class should be treated as a source of bean definitions. It lets you define beans and their configurations using methods annotated with `@Bean`.
- **@EnableAutoConfiguration:** The `@EnableAutoConfiguration` annotation triggers Spring Boot's auto-configuration mechanism. It automatically configures the application based on the classpath's dependencies and the content of the classpath. Spring Boot analyses the classpath, identifies the required libraries, and automatically configures various components, such as database connectivity, web servers, and messaging systems.
- **@ComponentScan:** The `@ComponentScan` annotation enables component scanning in the specified base packages. It instructs Spring to scan and detect

components, such as `@Component`, `@Controller`, `@Service`, and `@Repository`, to register them as beans in the application context.

- **Main application class:** The `@SpringBootApplication` annotation is typically used to annotate the main class of a Spring Boot application. This class serves as the entry point for the application and contains the `main()` method. By annotating this class, you enable Spring Boot features and ensure the application's necessary configurations and components are set up.

For example,

```
@SpringBootApplication
public class CodingNinjasAppApplication {

    public static void main(String[] args) {

        System.out.println("Welcome to the coding ninjas
application");
        ApplicationContext context =
SpringApplication.run(CodingNinjasAppApplication.class);

    }
```

The `@SpringBootApplication` annotation simplifies the configuration and setup of a Spring Boot application by combining multiple annotations into one. It enables auto-configuration and component scanning, making it easier to develop production-ready Spring Boot applications with minimal configuration.

I. YAML

YAML (*Yet Another Markup Language*) is a human-readable data serialisation format. It is commonly used for configuration files, data exchange, and defining structured data concisely and easily readable. YAML files use indentation and simple syntax to represent data structures, making the language more popular among developers and system administrators.

Here are some key features and characteristics of YAML:

- **Structure:** YAML uses indentation and nesting to represent data structures. It employs a hierarchical structure with indentation levels to define relationships between elements.
- **Readability:** YAML is designed to be human-friendly and easy to read. It uses plain text and avoids excessive punctuation, making it more natural and intuitive for users.
- **Data Types:** YAML supports various data types, including scalars (strings, numbers, booleans), sequences (arrays or lists), and mappings (key-value pairs or dictionaries). It provides a flexible and expressive way to represent structured data.
- **Comments:** YAML allows adding comments to the files using the '#' symbol. Comments provide additional explanations or annotations to the configuration and are ignored during parsing.
- **Inclusion and References:** YAML supports including and referencing other YAML files or parts of the same file. This feature enables code reuse and modularisation and simplifies complex configurations.
- **Serialisation and Deserialization:** YAML can be easily converted to and from programming language objects through serialisation and deserialisation processes. This allows developers to work with YAML data in their preferred programming language.
- **Portability:** YAML is language-agnostic and widely supported by various programming languages and frameworks. It provides a consistent format for data exchange and configuration format across different platforms and systems.

```
# Example YAML Configuration File

server:
  port: 8080
  host: localhost

database:
  driver: com.mysql.jdbc.Driver
  url: jdbc:mysql://localhost:3306/mydatabase
```

```
username: myusername
password: mypassword

logging:
  level: INFO
  file: logs/application.log
```

In the above example, the YAML file defines a server configuration with the port and host, a database configuration with driver, URL, username, and password, and a logging configuration with log level and log file path.

YAML's simplicity, readability, and flexibility make it popular for configuration files, data representation, and interchanging data between different systems.

J. Some extra annotations

- **@Service:** The `@Service` annotation marks a class as a service component. It is typically used to encapsulate business logic and perform operations related to the application's domain. Services are typically injected into other components, such as controllers, using the `@Autowired` annotation.
- **@Controller:** The `@Controller` annotation marks a class as a controller component in the MVC pattern. It handles incoming HTTP requests, performs required processing, and returns the appropriate response. Controllers typically contain methods annotated with `@RequestMapping` or other specialised request mapping annotations.
- **@Repository:** The `@Repository` annotation marks a class as a repository component. It is typically used for database access, data retrieval, and storage operations. Repositories often work with entities and provide an abstraction layer for data persistence. Repositories are typically injected into other components using the `@Autowired` annotation.
- **@Bean:** The `@Bean` annotation declares a method as a bean definition method in a configuration class. It tells Spring to manage the object's lifecycle returned by the annotated method and make it available for dependency injection. Beans can be configured with additional annotations such as `@Scope` or `@Qualifier` for customisation.
- **@RequestMapping:** The `@RequestMapping` annotation maps an HTTP request to a method in a controller class. It defines the URL pattern and the HTTP method(s) the method should handle. Additional annotations like `@GetMapping`, `@PostMapping`, etc., can be used for specific HTTP methods.

Conclusion

In conclusion, annotations play a crucial role in the Spring Boot framework, providing a declarative way to configure and manage various aspects of the application. Here's a summary of the key points regarding annotations in Spring Boot:

- **Configuration Annotations:** Spring Boot provides a range of annotations such as `@EnableAutoConfiguration`, `@ComponentScan`, and `@ConfigurationProperties` that simplify the configuration process and enable auto-configuration based on classpath scanning.
- **Dependency Injection Annotations:** Annotations like `@Autowired` and `@Qualifier` are used for dependency injection, allowing the framework to wire dependencies automatically and resolve ambiguities when multiple beans of the same type are available.
- **Stereotype Annotations:** Stereotype annotations such as `@Component`, `@Service`, `@Repository`, and `@Controller` are used to mark classes as Spring-managed components. These annotations help with component scanning and auto-detection of beans.
- **Lifecycle Annotations:** Annotations like `@PostConstruct` and `@PreDestroy` define lifecycle callback methods in managed beans. They allow you to execute specific tasks after bean initialisation or before destruction.

By utilising these annotations, developers can significantly reduce the amount of boilerplate code and configuration, resulting in more concise and readable code. Spring Boot's annotation-driven approach promotes convention over configuration and simplifies the development of robust and scalable applications.

Instructor Codes

- [Coding Ninjas Application](#)

References

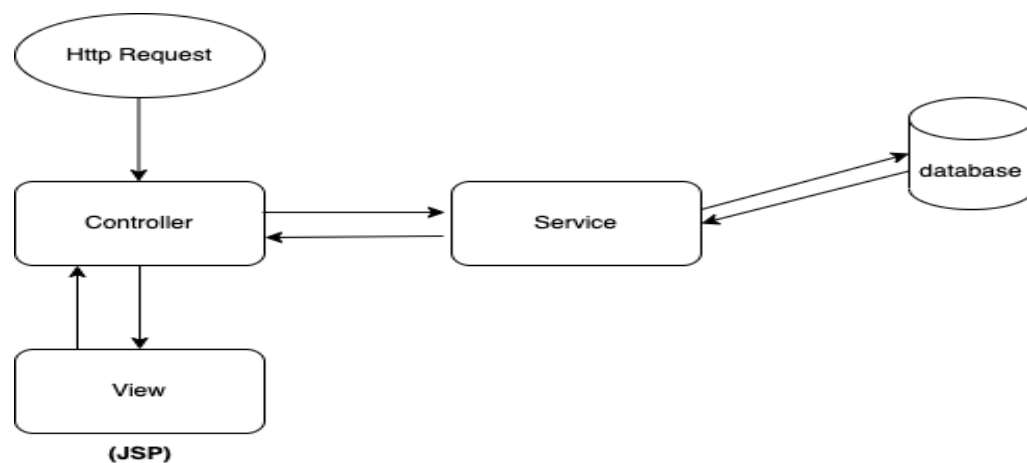
1. [Official Documentation](#)
2. [@Component Annotation](#)
3. [@Scope Annotation](#)
4. [Request and Session Scopes](#)
5. [@Qualifier Annotation](#)
6. [Spring lifecycle Annotation](#)
7. [@SpringBootApplication](#)
8. [YAML](#)
9. [More Annotations](#)
10. [@Data, @AllArgsConstructor, @NoArgsConstructor](#)
11. [Create your custom annotations](#)

Java Server Pages(JSP)

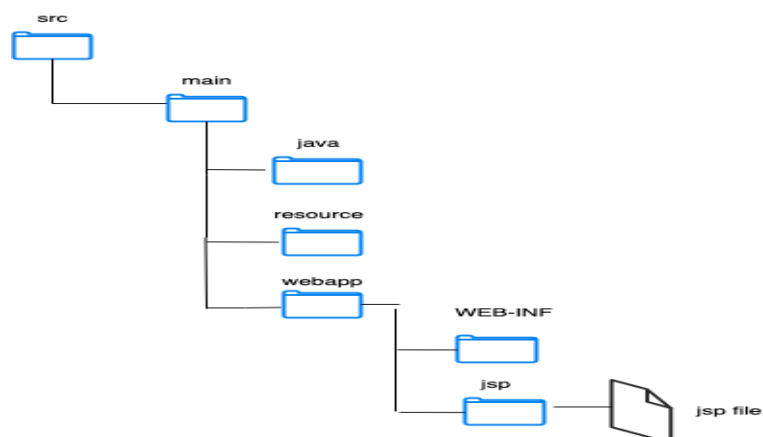
What Is JSP?

JSP (Java Server Pages) is a technology that allows developers to create dynamic web pages. It uses basic HTML tags to display and format the content. It will enable users to use specific JSP tags for Java code inside the HTML.

In a Spring Boot application, JSPs can be used as the view layer to display the content or take input from the user. For example, it can display data from a database or other sources or create forms allowing users to submit data to the server for processing.



Directory Structure of the JSP file:



Code Example:

- Jsp files act the same as HTML pages. You can write your HTML code as you want to do the task. An example is given below for the same:

```
<html>
<div>
  <h1>
    <a href="/signup">Sign me up</a>
  </h1>
</div>
</html>
```

- Here is another example of a ThankYou page in JSP:

```
<html>
<div>
  <h1>Thank You</h1>
</div>
</html>
```

To summarise, **JSP (JavaServer Pages)** is a technology that creates dynamic web pages in Java. It allows developers to embed Java code within HTML markup, enabling dynamic content generation. To use JSP in a Java web application, you must include the necessary dependencies, such as "**tomcat-embed-jasper**" and "**JSTL**", in your project configuration.

These dependencies provide the JSP rendering capabilities and the embedded Tomcat server. With JSP and the required dependencies, you can create JSP files, combine HTML with embedded Java code, and deploy the application to run the JSP pages on a server. Now let's look at the necessary dependencies required to use JSP.

Dependencies for JSP In Spring Boot

A. Jasper

The "**Tomcat Embed Jasper**" Spring Boot dependency supports JSP (JavaServer Pages) rendering within a Spring Boot application. It allows you to use JSP alongside other Spring Boot features as a view technology.

Here is a brief description of the relevant dependencies:

- **Tomcat-embed-jasper** is the main dependency providing JSP rendering capabilities within a Spring Boot application. It embeds the Tomcat Jasper JSP engine responsible for compiling and rendering JSP pages.
- **Tomcat-embed-core**: This dependency embeds the Apache Tomcat servlet container, Spring Boot's default container. It provides the necessary runtime environment for running JSP pages.

To use the "**Tomcat Embed Jasper**" dependency in your Spring Boot application, you need to include the following Maven dependency in your project configuration:

```
<dependency>
  <groupId>org.apache.tomcat.embed</groupId>
  <artifactId>tomcat-embed-jasper</artifactId>
</dependency>
```

You may also need to configure the "**application.yml**" file in your Spring Boot application to specify the JSP-related properties, such as the prefix and suffix for your JSP views. Here's an example:

```
spring:
  mvc:
    view:
      prefix: /WEB-INF/views/
      suffix: .jsp
```

You can include this application.yml file in your Spring Boot project's classpath, and Spring Boot will automatically read and apply these configuration properties when running your application.

B. JSTL

JSTL (JavaServer Pages Standard Tag Library) is a standard tag library that provides a set of tags for JavaServer Pages (JSP) to simplify the development of dynamic web applications. It is an integral part of the Java EE (Enterprise Edition) specification and is widely used in web applications built using JSP technology.

The JSTL library consists of several tag libraries, each serving a specific purpose. Here are some of the main JSTL tag libraries:

- **Core Tag Library:** This tag library provides tags for control structures (conditionals and loops), variable manipulation, and internationalisation support. It includes tags such as `<c:if>`, `<c:forEach>`, `<c:set>`, `<c:choose>`, etc.
- **Formatting Tag Library:** This library offers tags for formatting and presenting data. It includes labels such as `<fmt:message>` for retrieving localised messages, `<fmt:formatNumber>` for formatting numbers, `<fmt:formatDate>` for formatting dates, and more.
- **XML Tag Library:** This tag library provides tags for XML processing, allowing you to parse and transform XML data using JSTL. It includes tags such as `<x:parse>` for parsing XML, `<x:forEach>` for iterating over XML elements, and `<x:transform>` for applying XSLT transformations.
- **SQL Tag Library:** This tag library enables database access within JSP using SQL. It includes tags such as `<sql:setDataSource>` for defining a connection to a database, `<sql:query>` for executing SQL queries, `<sql:update>` for running SQL updates and more.

To use JSTL in your web application, include the JSTL library in your project's classpath and import the appropriate JSTL tag libraries into your JSP pages. You can typically include the JSTL library by adding the following Maven dependency:

```
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>jstl</artifactId>
  <version>1.2</version>
</dependency>
```

Once you have included the JSTL library, you can import the desired JSTL tag libraries in your JSP pages using the `<%@ taglib %>` directive. For example, to import the Core Tag Library, you would use the following directive:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

JSTL simplifies JSP development by abstracting everyday tasks into reusable tags, promoting separation of concerns and reducing the amount of Java code within JSP pages. It provides a standardised and consistent way to accomplish tasks in JSP-based web applications.

What is DevTools?

Spring Boot DevTools is a development-time module that provides several features to enhance the development experience when working with Spring Boot applications.

Here are some key features and functionalities provided by Spring Boot DevTools:

- **Automatic Restart:** DevTools monitors the classpath for changes and automatically restarts the Spring Boot application whenever a change is detected, eliminating the need to manually stop and start the application during development, significantly reducing turnaround time.
- **Hot Reloading:** DevTools supports hot reloading of specific resources, such as templates and static files, in addition to automatic restarts. Changes to these resources are immediately reflected in the running application without requiring a complete restart, allowing for faster iterative development.
- **Remote-Live Reload:** DevTools includes a remote live reload feature that allows you to trigger a browser refresh whenever changes are made to client-side resources. This feature is handy when working on front-end code and enables a seamless development experience between the backend and front end.
- **Developer-Friendly Error Page:** DevTools replaces the default Spring Boot error page with a more detailed and developer-friendly error page. It provides helpful information about the error, including the stack trace, request details, and application configuration.
- **Additional Developer Tools:** DevTools offers various tools and utilities, such as enhanced logging output, auto-configuration report, and property defaults report. These tools provide insights into the application's configuration and help diagnose potential issues during development.

To use Spring Boot DevTools in your Spring Boot project, you need to include the following Maven dependency in your project configuration:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-devtools</artifactId>
  <version>3.0.1</version>
</dependency>
```

References:

1. [JSP](#)
2. [JSTL](#)
3. [JSTL II](#)
4. [Devtools](#)

Note: We will explore them in detail in the upcoming lecture videos.

MVC Core Principle

Introduction to MVC:

MVC stands for Model-View-Controller; it is a design pattern that separates the application logic into three interconnected components: the model, the view, and the controller.

- **The Model** represents the data and the business logic of the application. It is responsible for handling the data and providing the necessary manipulation methods. It is also responsible for validating the data and ensuring that it is in the correct format.
- **The View** represents the user interface of the application. It is responsible for displaying the data to the user and providing a way for the user to interact with the application. It receives data from the model and displays it to the user in an appropriate format.
- **The Controller** is the component that receives the user's input and decides how to respond to it. It receives input from the view, processes it, and updates the model and the view accordingly. It acts as a mediator between the model and the view, controlling the data flow between them.

The main benefit of the MVC pattern is that it separates the concerns of the application into distinct components, making it easier to understand, maintain, and extend the application. It also promotes the reusability of the code, allowing different views to be used with the same model or vice versa.

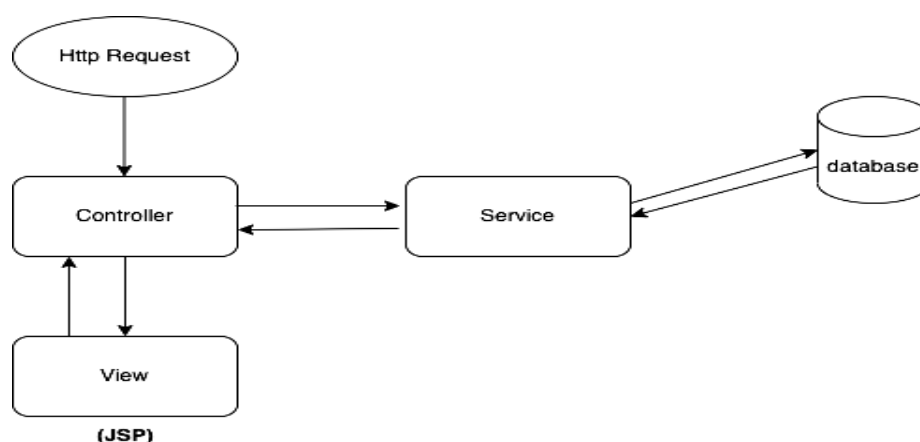


Figure: Diagram of how MVC Architecture works

Example:

A simple example of MVC is a basic calculator application. The Model would handle the calculations, such as addition and subtraction. The View would be the user interface, where the user can enter numbers and see the results. The Controller would receive the user's input and commands and then use the Model to perform the calculations and update the View to display the results.

In this example, the Model handles the data and calculations, the View handles the user interface, and the Controller coordinates and controls the interaction between the Model and the View. This separation of concerns makes the application easier to understand, maintain and extend.

Java Server Pages(JSP):

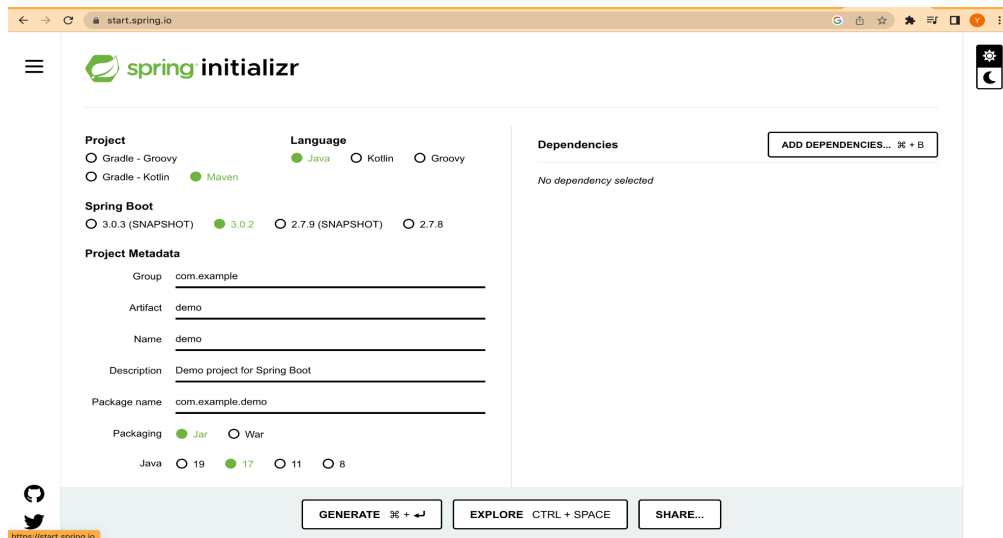
JSP (Java Server Pages) is a technology that allows developers to create dynamic web pages. It uses basic HTML tags to display and format the content. It will enable users to use specific JSP tags to use java code inside the HTML.

In a Spring Boot application, JSPs can be used as the view layer to display the content or take input from the user. For example, it can be used to display data from a database or other sources or to create forms that allow users to submit data to the server for processing.

Project Configuration:

A. Spring Initializer:

- Spring Initializer is a web-based tool provided by the Spring Framework that allows developers to quickly create a new Spring Boot project with minimal configuration. It can be accessed at start.spring.io.
- Using Spring Initializer, developers can create a new project by selecting the desired options and dependencies. The dependencies can be added to the project by selecting them from a list or manually entering their coordinates.
- Once the options and dependencies are selected, Spring Initializer generates a zip file containing the project structure, configuration files, and dependencies. Developers can then download the zip file and import the project into their preferred development environment.
- Spring Initializer is a convenient tool for quickly creating a new Spring Boot project with minimal configuration. It can save developers time by providing a simple way to configure a new project with the desired options and dependencies.



The screenshot shows the Spring Initializer web interface in a browser. The URL is start.spring.io. The interface is divided into several sections:

- Project:** Radio buttons for ☐ Gradle - Groovy, ☐ Gradle - Kotlin, and ☒ Maven.
- Language:** Radio buttons for ☒ Java, ☐ Kotlin, and ☐ Groovy.
- Spring Boot:** Radio buttons for ☐ 3.0.3 (SNAPSHOT), ☒ 3.0.2, ☐ 2.7.9 (SNAPSHOT), and ☐ 2.7.8.
- Project Metadata:** Text input fields for Group (com.example), Artifact (demo), Name (demo), Description (Demo project for Spring Boot), and Package name (com.example.demo).
- Packaging:** Radio buttons for ☒ Jar and ☐ War.
- Java:** Radio buttons for ☐ 19, ☒ 17, ☐ 11, and ☐ 8.
- Dependencies:** A section with the text "No dependency selected" and a button "ADD DEPENDENCIES...".

At the bottom, there are three buttons: "GENERATE" (with a download icon), "EXPLORE" (with a keyboard shortcut "CTRL + SPACE"), and "SHARE...".

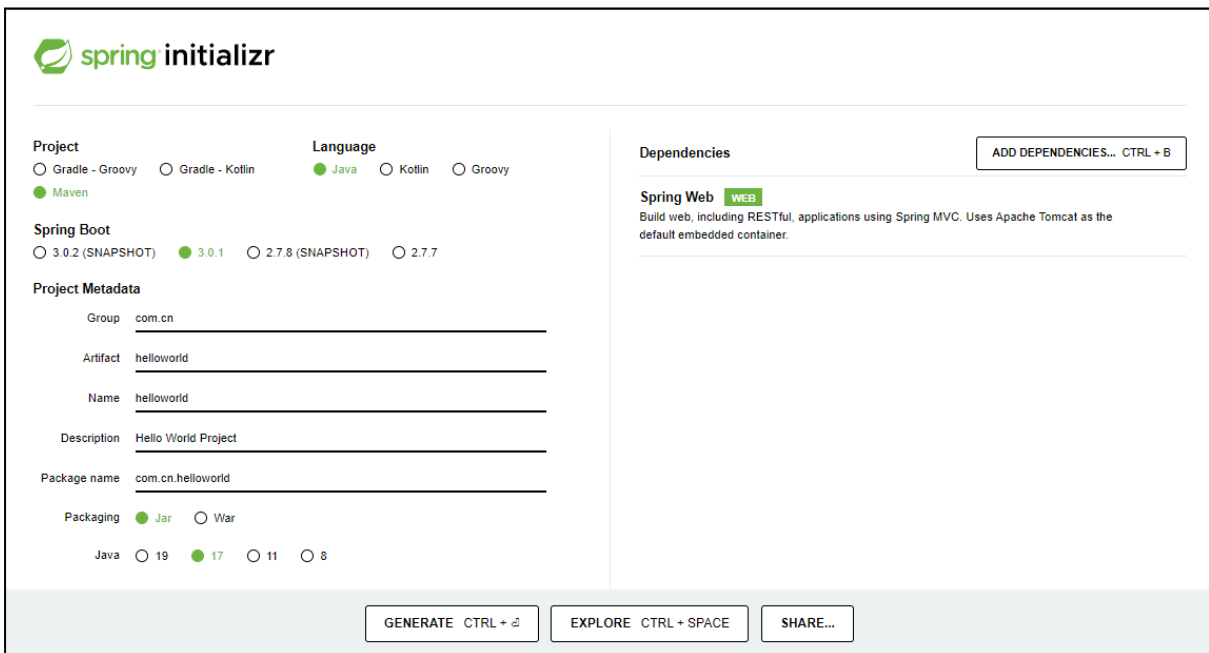
B. Dependencies:

1. **"Web" dependency:** In a Spring Boot project, the "web" dependency is a module that provides support for building web applications using Spring MVC (Model-View-Controller). It includes various features, such as support for handling HTTP requests and responses, form submissions, and exceptions.
2. **"Devtools" dependency:** The "dev tools" dependency in a Spring Boot project provides development-time support for developers, including features such as automatic restart, live reload, remote development, and improved logging. These features make development easier and more efficient. It is typically used in development environments and is not recommended for production environments.

3. **JSTL:** JSTL (JavaServer Pages Standard Tag Library) is a set of tags that can be used in JSP pages to perform common tasks such as iteration, conditional logic, and URL handling. Including the "jstl" dependency in the Spring Boot project gives access to these tags and makes it easy to create dynamic web pages by separating the presentation logic from the business logic.

C. Getting started with the spring boot project:

1. Visit <https://start.spring.io> (This is a web-based Spring Initializer).
Add Spring Web Dependency to your project.



The screenshot shows the Spring Initializr web interface. On the left, under 'Project', 'Maven' is selected. Under 'Language', 'Java' is selected. Under 'Spring Boot', '3.0.1' is selected. The 'Project Metadata' section has fields for Group (com.cn), Artifact (helloworld), Name (helloworld), Description (Hello World Project), and Package name (com.cn.helloworld). Under 'Packaging', 'Jar' is selected. Under 'Java', '17' is selected. On the right, under 'Dependencies', 'Spring Web' is selected with a 'WEB' tag. At the bottom, there are buttons for 'GENERATE CTRL + G', 'EXPLORE CTRL + SPACE', and 'SHARE...'.

In the end, the selections should look like this, and then click on **GENERATE** button on the bottom. It would download a zip file.

2. Extract the zip in a folder.
3. Import the project in Eclipse by going to -
File -> Open projects from File System -> Directory -> Select the extracted folder -> Finish.
4. Add the following dependencies in **pom.xml**.

```
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>jstl</artifactId>
  <version>1.2</version>
</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-devtools</artifactId>
</dependency>
<dependency>
  <groupId>jstl</groupId>
  <artifactId>jstl</artifactId>
  <version>1.2</version>
</dependency>
```

D. Configuration for Application.yml file:

1. For the port number.
2. For the JSP file, add prefixes and suffixes.

```
spring:
  mvc:
    view:
      prefix: "WEB-INF/jsp/"
      suffix: ".jsp"
server:
```

Controller Class:

A Controller class in Spring Boot is a Java class that acts as an intermediary between a client and the server to handle HTTP requests. It handles incoming HTTP requests, processes the data, and returns a response to the client. Controllers are annotated with the `@Controller` annotation and contain methods annotated with `@RequestMapping` to map to specific URLs.

1. HomeController

```
@Controller
public class HomeController {
    @RequestMapping("/home")

    public String getHomePage(){
        return "home";
    }
}
```

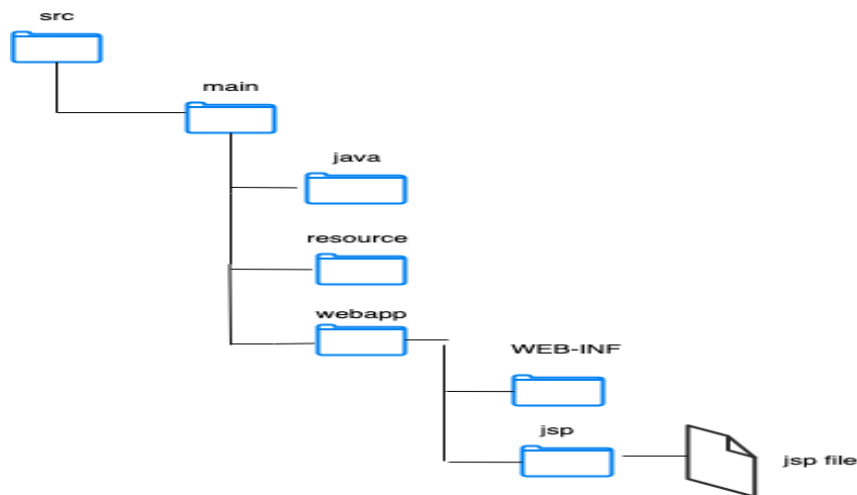
2. SignupController

```
@Controller
public class SignupController {
    @RequestMapping("/signup")
    public String signup(){
        return "signup";
    }
}
```

View Page(JSP pages):

In a Spring Boot application, JSPs can be used as the view layer to display the content or take input from the user. For example, it can be used to display data from a database or other sources or to create forms that allow users to submit data to the server for processing.

A. Directory Structure of the JSP file:



B. Add two pages:

These pages will be triggered using the controllers.

Welcome.jsp

```

<html>
<div>
  <h1>
    <a href="/signup">Sign me up</a>
  </h1>
</div>
</html>

```

Signup.jsp

```

<html>
<div>
  <h1>
    This is signup Page.
  </h1>
</div>
</html>

```


Domain layer

The Domain layer in Spring Boot is a crucial component in the architecture of a Spring Boot application. It contains the domain objects, which are the classes that represent the business entities and their relationships.

1. User interface:

```
public interface User {  
    public boolean createUser(String name,String gender,String location,String  
college);  
    public Integer saveUser();  
  
}
```

2. StudentUser Class:

We have to implement the User interface for better code reusability.

```
package com.example.website.domain;  
  
import org.springframework.stereotype.Component;  
  
@Component  
public class StudentUser implements User{  
    String name;  
    String gender;  
    String location;  
    String college;  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public String getGender() {  
        return gender;  
    }  
  
    public void setGender(String gender) {  
        this.gender = gender;  
    }  
  
    public String getLocation() {
```

```
        return location;
    }

    public void setLocation(String location) {
        this.location = location;
    }

    public String getCollege() {
        return college;
    }

    public void setCollege(String college) {
        this.college = college;
    }

    @Override
    public boolean createUser(String name, String gender, String location, String
college) {
        this.name=name;
        this.gender=gender;
        this.location=location;
        this.college=college;
        return true;
    }

    @Override
    public Integer saveUser() {
        System.out.println(this.name);
        return 0;
    }
}
```

Service layer

The Service layer in Spring Boot is a component in the architecture of a Spring Boot application that acts as an intermediary between the Domain layer and the Presentation layer. The Service layer contains the application's business logic and is responsible for processing requests, using the objects in the Domain layer to perform the necessary actions.

1. UserService interface:

```
public interface UserService {  
    public User getUser();  
    public boolean signUp(String name,String gender,String location,String  
college);  
}
```

2. StudentUserService Class:

```
@Service  
public class studentUserService implements UserService{  
  
    @Autowired  
    User studentUser;  
    @Override  
    public User getUser() {  
        return studentUser;  
    }  
  
    @Override  
    public boolean signUp(String name, String gender, String location, String  
college) {  
        boolean isStudentCreated=studentUser.createUser(name, gender, location,  
college);  
        studentUser.saveUser();  
        return isStudentCreated;  
    }  
}
```

MODEL AND MODEL ATTRIBUTE

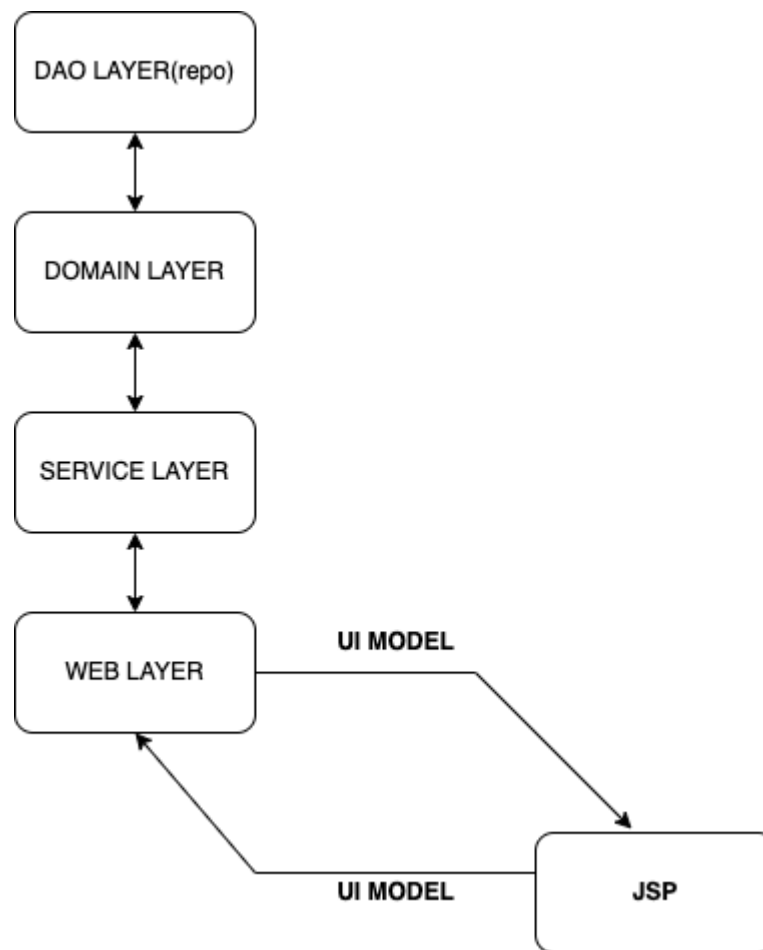


Figure: Representation of WEB Layer and JSP connection using MODEL

Model and model attribute are important concepts in Spring UI that are used to bind data to a view in a web application.

A model refers to an object that holds the data that will be displayed on a view. This data can be any type you want to pass from the controller to the view, such as lists, maps, or simple strings.

The model is created in the controller and can be accessed in the view using the JSTL tags.

A model attribute refers to an annotated parameter in a controller's method. It binds the form data to an object that can be passed to the view. For example, if you have a form that captures user information, you can use a model attribute to bind the form data to an object that represents the user, such as a User object. The model attribute is then passed to the view and can be accessed using JSTL tags.

To use the model and model attribute in a Spring UI application, you need to create a model in the controller, add the model attribute to a method in the controller, and use JSTL tags to access the data in the view.

This code needs to be added in the **User controller** class

```
@Autowired
UserService userService;

@RequestMapping(value = "/signUp")
public String getSignupPage(Model ui) {
    User user = userService.getUser();
    ui.addAttribute("user", user);
    return "signup";
}
```

MVC FORM TAG LIBRARY

The MVC (Model-View-Controller) form tag library is a set of custom JSP tags that provide a convenient way to create dynamic HTML forms in a JSP-based web application that follows the MVC design pattern.

The tag library provides tags for generating various types of form controls, such as text fields, radio buttons, checkboxes, select lists, and more. In an MVC-based web application, the form tag library is used in the View component, which is responsible for presenting the data to the user.

The tags provided by the form tag library are used to render the form controls. The form data is automatically populated from the Model component, which is responsible for managing the application data.

The form tag library provides several benefits over traditional HTML forms, including

Abstraction: It provides a higher level of abstraction, making it easier for developers to create forms without having to write complex HTML code.

Data binding: The form tag library provides data binding between the Model component and the form controls, automatically populating the form with data from the Model and updating the Model with data from the form.

Validation: The form tag library provides built-in support for form validation, making it easy to validate form data and display error messages to the user. Overall, the MVC form tag library is an essential component of MVC-based web applications, providing a powerful and flexible solution for creating dynamic HTML forms.

Configuration:

To utilize the form tag library in a JSP page, it's necessary to add a reference to the required configuration. This is done by including the following directive at the start of the JSP file:

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
```

This directive maps the form prefix to the URI of the Spring Framework form tag library, allowing you to use the form tags in your JSP code.

In JSP, several form-related tags can be used to create dynamic HTML forms, including

form:form	This tag creates an HTML form and binds it to a model object. It can be used to specify the action URL, the HTTP method, and other form attributes.
form:input	This tag creates an HTML text input field and binds it to a model property. It can specify the input type (such as text, password, or hidden), the size, the max length, and other attributes.
form:password	This tag creates an HTML password input field and binds it to a model property. It can specify the size, the max length, and other attributes.
form:textarea	This tag creates an HTML textarea field and binds it to a model property. It can specify the rows, columns, and other attributes.
form:select	This tag creates an HTML select list and binds it to a model property. It can specify the options, the selected value, and other attributes.
form:checkbox	This tag creates an HTML checkbox and binds it to a model property. It can specify the selected state and other attributes.
form:radio	This tag creates an HTML radio button and binds it to a model property. It can be used to specify the selected state and other attributes.

Now after Understanding the Form tags let's see the sample code for the "Signup.jsp."

Signup.jsp

```
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<h1>sign up Page</h1>
<form:form action="registerUser" modelAttribute="user">
```

```

name<form:input path="name" />
<br />
<br />
Gender
<br />
Male<form:radiobutton path="gender" value="male" />
Female<form:radiobutton path="gender" value="female" />
<br/>
<br/>
    Location:
<form:select path="location">
    <form:option value="India"></form:option>
    <form:option value="NRI"></form:option>
</form:select>
<br />
<br />
    College
<form:select path="college">
<form:option value="test123"></form:option>
<form:option value="testCollege"></form:option>
</form:select>
    <br />
    <br />

    <input type="submit" />
</form:form>
</html>

```

To get the details from the form, the following code needs to be added in the **User controller** to register the user:

```

@RequestMapping(value="/register")
public String getResposePage(@ModelAttribute("user") StudentUser studentUser) {

    if(userService.signup(studentUser.getName(),studentUser.getGender(),studentUser.
        getLocation(),studentUser.getCollege())) {
        return "success";
    }
    return "signup";
}

```

The domain layer, controller layer, and service layer are all important components in a software application architecture.

The domain layer is responsible for defining and managing the core business logic and rules, while the controller layer acts as an intermediary between the domain layer and the presentation layer, handling user requests and input validation.

The service layer is a higher level of abstraction than the domain layer, it defines and implements the application's core services that can be used by multiple parts of the

application, including the domain and controller layers. This layer helps to encapsulate complex logic, promote reusability, and decouple different parts of the application.

In conclusion, these three layers work together to provide a well-structured and modular architecture for a software application, allowing for a clear separation of concerns, better maintainability, and easier testing.

Now we are going to understand how where to store the data

DAO layer

The Data Access Object (DAO) layer is a design pattern that is often used in conjunction with the Model-View-Controller (MVC) architecture. It provides a way to interact with the data storage in a separated and decoupled manner. The DAO layer acts as an intermediary between the Model layer and the data storage.

1. DAO Interface

```
package com.example.registration.repository;

import java.util.Optional;

public interface DAO<T> {
    public Optional<T> get(Integer id);
    int save(T t);
}
```

2. StudentUserDao Class

```
package com.example.registration.repository;

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

import org.springframework.stereotype.Component;
import org.springframework.stereotype.Repository;

import com.example.registration.domain.StudentUser;

@Repository
public class StudentUserDAO implements DAO<StudentUser>{
    private List<StudentUser> studentUserList = new ArrayList<>();

    @Override
    public Optional<StudentUser> get(Integer id) {
```



```
        return Optional.ofNullable(studentUserList.get(id));
    }

    @Override
    public int save(StudentUser studentUser) {
        int studentId = studentUserList.size();
        studentUser.setUserId(studentId);
        studentUserList.add(studentUser);
        System.out.println("user saved");
        return studentId;
    }
}
```

Conclusion

The Model-View-Controller (MVC) architecture in Spring Boot consists of three main layers: Model, View, and Controller. The Model layer represents the data and business logic of the application. The View layer is responsible for presenting the data to the user.

The Controller layer acts as an intermediary between the Model and View layers, handling user inputs and directing data flow between the two.

The MVC architecture provides a clear separation of concerns, making it easier to maintain and modify the code in the future. Additionally, Spring Boot provides a robust framework for building and deploying applications using the MVC pattern.

Instructor Codes

- [Website Application](#)

References

1. [MVC Architecture](#)
2. [MVC Architecture II](#)
3. [JSP](#)
4. [JSP Tag Library](#)

MVC and Web

Query Parameters:

In Spring Boot, query parameters pass data from the client to the server through the URL. Query parameters are appended to a URL's end and preceded by a question mark. Each query parameter consists of a key-value pair, separated by an equals sign, and an ampersand separates multiple parameters.

Here's an example of a URL with query parameters:

```
http://example.com/search?q=spring+boot&sort=rating
```

In this example, the query parameter "q" has a value of "spring boot", and the query parameter "sort" has a value of "rating".

In a Spring Boot application, you can easily retrieve query parameters using the `@RequestParam` annotation. For example, let's say you have a REST endpoint that accepts query parameters for a search:

```
@GetMapping("/search")
public String search(@RequestParam("q") String query,
    @RequestParam("sort") String sortBy) {
    // Search logic here
    return "Search results for " + query + ", sorted by " + sortBy;
}
```

In this example, the search method accepts two query parameters: "q" and "sort". The values of these parameters are retrieved using the `@RequestParam` annotation and are passed as arguments to the method. The method then returns a string containing the search results, including the query and sort parameters.

When this endpoint is accessed with a URL like `http://example.com/search?q=spring+boot&sort=rating`, the search method will be called with the values "spring boot" for the query parameter and "rating" for the sort parameter.

Model VS ModelMap

- "Model" and "ModelMap" are both classes used in the Spring Boot framework for Java to represent data that can be used to render views in web applications.
- The main difference between the two is that Model is an interface that defines a contract for objects that can be used to pass data between a controller and a view. It's a simple key-value store that allows you to store and retrieve values using a string key.
- On the other hand, ModelMap is a concrete implementation of the Model interface and provides additional functionality to store and manipulate data. It is also the default implementation used by Spring Boot when passing data between a controller and a view.
- In practice, both Model and ModelMap can be used interchangeably. However, ModelMap offers more advanced features, such as the ability to set a default value for a key not present in the map or copy the contents of one ModelMap to another.

Here's an example of how to use Model and ModelMap in a Spring Boot controller:

```
@RequestMapping("/example")
public String example(Model model) {
    model.addAttribute("message", "Hello World!");
    return "example";
}

@RequestMapping("/example2")
public String example2(ModelMap model) {
    model.addAttribute("message", "Hello World!");
    return "example";
}
```

In both cases, the controller is adding a "message" attribute to the model or model map, which can then be used by the view to display the message. The only difference is that the first example uses the Model interface, while the second example uses the ModelMap implementation.

Support Page:

After registering the User we need to create a new page called the support page, and to do that, we need a support controller and support JSP file. We need to have a User id on that support page. We will see all these in the coming path with the following concept.

SignUpController

```
@RequestMapping("/registerUser")
public String createUser(@ModelAttribute(value = "user") StudentUser
studentUser) {
    int      userId      =      userService.signUp(studentUser.getName(),
studentUser.getGender(),studentUser.getLocation(), studentUser.getCollege());
    if(userId != -1)
    {
        ModelAndView modelAndView =new ModelAndView("redirect:welcome?id="+userId);
        return modelAndView.getViewName();
    }
    return "signup";
}
@RequestMapping("/welcome")
public String showWelcomePage(@RequestParam("id") String userID,ModelMap map) {
    map.addAttribute("userID",userID);
    return "welcome";
}
```

Support Controller

```
@RequestMapping("/support")
    public String getSupportPage(@RequestParam("id") String id,ModelMap
map){
    map.addAttribute("message","Welcome to support page "+id);
    return "support";
}
}
```

Welcome.jsp

```
<html>
<h1>Thank you, you are now a member</h1>
<a href="/support?id=${userID}">Contact support</a>
<a href="/instructors">my Instructors</a>
</html>
```

Support.jsp

```
<h1>Support Page</h1>
<div>${message}</div>
<% String a = "rahul is using java here"; %>
<%=a%>
```

Model vs ModelAndView vs ModelMap

Here's a brief overview of each:

- **Model:** Model is an interface that provides a simple way to pass data to a view. It's a key-value store that allows you to store and retrieve values using a string key. The Model interface is implemented by the controller and passed as a parameter to the controller method. The values set in the Model object are made available to the view.
- **ModelAndView:** ModelAndView is a class that combines a model with a view name. It provides more flexibility than the Model interface, allowing you to set both the model and the view name in a single object. ModelAndView can also add information to the response, such as HTTP headers or a status code.
- **ModelMap:** ModelMap is a class that implements the Model interface and provides additional functionality. It's similar to a Map and allows adding and retrieving values using a string key. ModelMap provides additional convenience methods, such as the ability to set a default value for a key not present in the map.

In practice, all three classes can pass data to a view, but the choice between them depends on the application's requirements. If you need to pass data to a view and specify the view name separately, you can use ModelAndView. If you just need to pass data to a view, you can use Model. If you need to add additional functionality to the model, such as setting a default value or copying the contents of one model to another, you can use ModelMap.

Path param

A path parameter, also known as a path variable, is a variable that is part of a URL path in a web application. In the context of RESTful web services, path parameters are used to identify a specific resource.

In a RESTful web service, the URL is used to identify a resource, and path parameters provide additional information about the resource. For example, consider a URL that represents a user profile:

```
https://example.com/users/{userId}.
```

In this URL, {userId} is a path parameter that identifies the specific user profile to retrieve. When a request is made to this URL with a specific userId, the server retrieves the data for that user profile and returns it in the response.

Instructor Page:

Signup Controller

```
@RequestMapping("/instructors")
public String showInstructorsPage(ModelMap map) {
    //creating instructors using hash maps
    HashMap<String, Object> instructor1 = new HashMap<String, Object>();
    instructor1.put("name", "Rahul Mohan");
    instructor1.put("age", 23);
    instructor1.put("id", 243);

    //ArrayList for a list of instructors
    ArrayList<HashMap<String, Object>> listOfInstructors = new
    ArrayList<HashMap<String, Object>>();
    listOfInstructors.add(instructor1);

    // sending key value pairs to the view
    map.addAttribute("instructors", listOfInstructors);
    return "instructors";
}
```

Instructor.jsp

```
<%@page import="java.util.HashMap"%>
<%@page import="java.util.ArrayList"%>
<html>
```

```
<h1>Here are you instructors:</h1>
<%
    ArrayList<HashMap<String,Object>> listOfInstructors =
        (ArrayList<HashMap<String,Object>>)request.getAttribute("instructors");
    for(HashMap<String,Object> instructor: listOfInstructors) {
%>
name <%=instructor.get("name")%>
<a href="profile/<%=instructor.get("id")%>">profile</a>
<%
    }
%>
</html>
```

Profile page:

Now, adding the profile page

SignupController

```
@RequestMapping("/profile/{profileID}")
public String ShowProfile(@PathVariable("profileID") String profileID,ModelMap
map) {
    map.addAttribute("profileID",profileID);
    return "profile";
}
```

Profile.jsp

```
<html>
    <h1>Profile Page</h1>
    <div>id from path param:${profileID}</div>
</html>
```


Conclusion

In conclusion, Query Parameters and Path Parameters are two important ways of passing data in a URL for a web application. Query Parameters are used for non-essential data that doesn't change the output of the endpoint, while Path Parameters are used for essential data that alters the output.

A Model is an object that holds data and is used by the controller to interact with the view. It separates the concerns of business logic and presentation logic. ModelAndView and ModelAndView are two popular approaches for returning data from a controller to a view in Spring MVC.

ModelView is a simple object that contains both the data model and the view name, while ModelAndView contains both the model and view information as separate properties. Both approaches provide flexibility in designing and structuring web applications with Spring MVC.

Overall, understanding the differences and use cases of Query Parameters, Path Parameters, Model, ModelView, and ModelAndView is crucial for building efficient and scalable web applications.

Instructor Codes

- [Website Application](#)

References

1. [Request Param](#)
2. [Model VS ModelView VS ModelAndView](#)

Understanding Web Services

Introduction

We encountered terms like APIs and web services in the last few videos. While these terms are related to software development and integration, they have distinct meanings and purposes. This documentation will explore some of the crucial differences between **APIs and Web services**, **REST API and SOAP** and provide simple explanations and examples to help you understand them better.

Web Services

A. What is web Service?

Web services are APIs designed to be accessed online using standard web protocols. Web services enable different software systems to communicate and exchange data regardless of the technologies or platforms they use.

Example: A Restaurant Web Service

Imagine your favourite restaurant that wants to offer online services. Customers can:

1. Look at the menu.
2. Place an order.
3. Check the status of their order.
4. Find out the restaurant's location and opening hours.

How It Works:

1. Viewing the Menu

When you want to see the restaurant's menu on your phone or computer:

- **Your Action:** You click a button to view the menu.
- **What Happens:** Your device sends a request to the restaurant's web service asking for the menu.
- **Result:** The web service sends back the menu, which appears on your screen.

2. Placing an Order

When you decide to place an order:

- **Your Action:** You select items from the menu and submit your order.

- **What Happens:** Your device sends the order details to the restaurant's web service.
- **Result:** The web service confirms your order and gives you an order number.

3. Checking Order Status

To find out how your order is progressing:

- **Your Action:** You check the order status on your device.
- **What Happens:** Your device asks the web service for the current status of your order using your order number.
- **Result:** The web service replies with the status (e.g., being prepared, ready for pickup).

4. Getting Restaurant Information

To find out the restaurant's location and hours:

- **Your Action:** You click to view the restaurant's info.
- **What Happens:** Your device sends a request to the web service for this information.
- **Result:** The web service sends back the address and hours, which are displayed on your screen.

HTTP and HTTPS

HTTP (HyperText Transfer Protocol): This is like the language your device and the restaurant's web service use to talk to each other. It defines how messages are sent and received.

HTTPS (HyperText Transfer Protocol Secure): This is a secure version of HTTP. It ensures that the information exchanged between your device and the web service is private and safe from hackers.

Benefits of Web Services

- **Convenience:** You can easily access the restaurant's services from your phone or computer.
- **Real-Time Updates:** You get immediate updates on your order status.
- **Easy Access:** Quickly find important information like the menu and restaurant hours.
- **Security:** HTTPS keeps your personal information, like payment details, secure.

Conclusion

Web services make it possible for you to interact with a restaurant online smoothly and securely. Whether you're viewing the menu, placing an order, checking your order status, or finding out the restaurant's location, web services handle the

communication between your device and the restaurant's system, ensuring everything works efficiently.

B. Data Representation in web services

- Web services use XML (eXtensible Markup Language) to structure and represent data. XML allows developers to define custom markup tags and hierarchies to organise data in a structured manner. It is widely adopted and provides a human-readable format that software applications can quickly parse.
- JSON (JavaScript Object Notation) is also commonly used for data representation in web services, particularly in RESTful APIs. The choice between XML and JSON depends on specific requirements, technology stack, and developer preferences.

C. Architectural styles

There are two main architectural styles or approaches used in web services:

- I. Representational State Transfer (REST)
- II. Simple Object Access Protocol (SOAP)

We will explore the two architectural styles later in the document.

XML (eXtensible Markup Language)

What is XML?

XML is a markup language designed to store and transport data. It is both human-readable and machine-readable, which makes it versatile for various applications.

Key Features:

1. Structure: XML documents are composed of nested elements enclosed in tags. For example:

```
<person>
  <name>John Doe</name>
  <age>30</age>
  <address>
    <street>Main Street</street>
    <city>Metropolis</city>
  </address>
</person>
```

2. Self-Descriptive: XML is designed to be self-descriptive; the structure and the meaning of data are clear.
3. Extensibility: You can define your own tags, making it very flexible.
4. Validation: XML supports DTD (Document Type Definition) and XSD (XML Schema Definition) for validating the structure and content.
5. Hierarchical: XML inherently supports a hierarchical structure, making it suitable for representing complex nested data.

JSON (JavaScript Object Notation)

What is JSON?

JSON is a lightweight data interchange format, easy for humans to read and write, and easy for machines to parse and generate. It was initially derived from JavaScript but is now language-independent.

Key Features:

1. Structure: JSON data is represented as key-value pairs. For example:

```
{  
  "name": "John Doe",  
  "age": 30,  
  "address": {  
    "street": "Main Street",  
    "city": "Metropolis"  
  }  
}
```

2. Simplicity: JSON is simpler and more compact than XML. It uses arrays and objects to represent data structures.
3. Data Types: JSON supports a limited set of data types: strings, numbers, objects, arrays, true/false, and null.
4. Interoperability: JSON is easy to parse and generate in most programming languages.

XML vs. JSON

Format and Readability:

XML: Verbose and more human-readable due to descriptive tags. Better suited for documents where the structure and hierarchy are crucial.

JSON: More compact and easier to read and write for simple data structures. Ideal for data interchange.

Data Handling:

XML: Better suited for complex data with mixed content and hierarchical structures. Supports comments.

JSON: Ideal for simple to moderately complex data. Easier to parse and process programmatically.

Metadata and Self-Describing Nature:

XML: Provides more explicit metadata through tags and attributes, making it very self-descriptive.

JSON: Relies on the structure and key names to convey metadata, which can be less explicit than XML.

Extensibility and Standards:

XML: Highly extensible with a range of standards like XSLT, XPath, and XQuery for transforming and querying XML data.

JSON: Simpler, with fewer formal standards, but integrates well with JavaScript and modern web APIs.

Performance:

XML: Parsing and serialisation can be slower due to its verbosity and complexity.

JSON: Generally faster to parse and serialize because of its simplicity and compactness.

Validation:

XML: Robust validation options with DTD and XSD, ensuring strict adherence to data schemas.

JSON: Schema validation is possible with JSON Schema but is less mature compared to XML's validation mechanisms.

Conclusion

XML is better suited for applications requiring complex data structures, extensive validation, and explicit metadata, while JSON is preferred for its simplicity, efficiency, and ease of use in web applications and data interchange. The choice between XML and JSON will depend on your specific use case and requirements.

REST API vs SOAP

In web services, two popular protocols are often used for communication between applications: REST API and SOAP. These protocols define the rules and formats for exchanging data over the internet. In this article, we'll explore the differences between REST API and SOAP, using simple language and examples to help you grasp the concepts.

REST API (Representational State Transfer)

REST API is an architectural style that uses HTTP as its communication protocol. It focuses on exposing resources (such as data or functionality) as URLs, known as endpoints. These endpoints represent different operations that can be performed on the resources.

1. **Simplicity and Lightweight:**

REST API is designed to be simple and lightweight, making it easy to understand and implement. It uses standard HTTP methods like GET, POST, PUT, and DELETE to perform actions on resources. For example, you can request an HTTP GET request to the appropriate endpoint to retrieve data from a REST API.

2. **Stateless Communication:**

REST API follows a stateless communication model, meaning each request from a client to a server contains all the necessary information to be understood. The server doesn't keep track of the client's previous interactions. This simplifies the server implementation and allows for scalability.

3. **Data Formats:**

REST API commonly uses JSON (JavaScript Object Notation) or XML (eXtensible Markup Language) as data formats for request and response payloads. JSON is more widely used due to its simplicity and compatibility with modern web technologies.

SOAP (Simple Object Access Protocol)

SOAP protocol uses XML to structure the data being exchanged between applications. It relies on rules for defining the message format and communication details.

1. **Complex and Robust:**

SOAP is more complex than REST API. It provides a standardised message structure definition, including request and response formats. SOAP also offers features like encryption, security, and error handling, making it suitable for enterprise-level applications.

2. **Communication Protocol:**

Unlike REST API, SOAP is not limited to HTTP. It can use HTTP, SMTP, or JMS (Java Message Service). This flexibility allows SOAP to be used in various communication scenarios.

3. XML-based Messaging:

SOAP messages are typically formatted using XML. This structured format ensures compatibility across different platforms and programming languages. SOAP defines a specific message envelope structure, including headers and body elements.

Example - REST API vs SOAP

Let's compare REST API and SOAP based on an example of retrieving customer information from a server.

REST API:

- Endpoint: GET /customers/{id}
- Request: A GET request to the specific URL endpoint with the customer ID in the path.
- Response: The server responds with a JSON or XML representation of the customer data containing the requested information.
- Example Request: GET /customers/123
- Example Response:

```
{
  "id": 123,
  "name": "John Doe",
  "email": "johndoe@example.com",
  "address": "123 Main St"
}
```

SOAP:

- Endpoint: A SOAP message is sent to the server's SOAP endpoint.
- Request: A SOAP request message containing the customer ID as a parameter.
- Response: The server sends a SOAP response message encapsulating the requested customer information.
- Example Request:

```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:web="http://example.com/webservice">
  <soapenv:Header/>
  <soapenv:Body>
    <web:GetCustomerRequest>
      <web:CustomerId>123</web:CustomerId>
    </web:GetCustomerRequest>
  </soapenv:Body>
</soapenv:Envelope>
```

- In this example, the REST API uses a simple HTTP GET request to retrieve customer information by specifying the customer ID in the URL. The server responds with a JSON or XML representation of the customer data.
- On the other hand, the SOAP-based approach involves constructing a SOAP request message with the customer ID as a parameter and sending it to the server's SOAP endpoint. The server then sends a SOAP response message encapsulating the requested customer information.

Overall, the REST API approach is more lightweight and uses standard HTTP protocols and data formats like JSON, while SOAP involves more complex XML-based messaging and additional protocols.

REST is favoured for simplicity and ease of use, whereas SOAP is often used in enterprise scenarios requiring more advanced features like security and transactions.

Conclusion

In conclusion, we have explored APIs, web services, and the differences between REST API and SOAP. A few of the inferences we have drawn are as follows:

- APIs act as a communication bridge between software applications, while web services are APIs accessed online using web protocols. Web services commonly use XML or JSON for data representation.
- REST API follows the principles of simplicity and lightweight, using HTTP methods and JSON for data exchange. Conversely, SOAP is more complex and robust, supporting various protocols and using XML for messaging.
- The choice between REST API and SOAP depends on specific requirements. REST API is preferred for its simplicity and compatibility with modern technologies, while SOAP offers advanced features suitable for enterprise-level applications.

Understanding these concepts helps developers make informed decisions for designing and integrating software systems, ensuring efficient communication between applications.

APIs - Application Programming Interfaces

What is API?

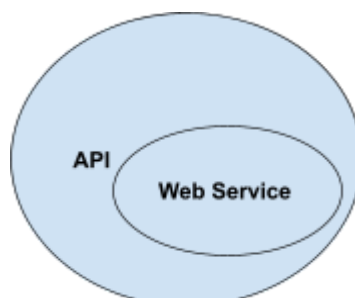
- API stands for **Application Programming Interface**. In simple terms, an API is a set of rules and protocols that allows different software applications to communicate and interact with each other. It acts as a bridge between two applications, enabling them to exchange data and functionality seamlessly.
- APIs define the methods and formats to request and receive information between applications. They provide a standardised way for developers to access specific features or services offered by another application or system. Think of APIs as a messenger that helps different applications talk to each other and share information.

API vs Web Service via Example

Let's say you have a mobile application that displays the current weather. You must fetch weather data from a service provider to show it in your app.

- **API example:** In this scenario, you use the weather service provider's API. The API provides a set of functions or endpoints that allow you to make specific requests for weather data. You would request the API for the current weather of a particular location, and the API would respond with the relevant data in a structured format, such as JSON. You can then extract and display the weather information in your mobile app. The API acts as the intermediary, enabling your app to communicate and retrieve data from the weather service.
- **Web service example:** In this case, the weather service provider offers a web service for accessing weather information online. You would request an HTTP to the web service's specific URL and any required parameters (e.g., location). The web service would process your request and respond with the weather data in a standardised format like XML or JSON. You can then parse the response and display the weather information in your app. In this case, the web service is an API accessed via web protocols.

The critical difference lies in the underlying technologies and protocols used. An API is a general concept for facilitating communication between applications. At the same time, a web service is a specific type of API accessed over the web using web-based protocols like HTTP. Hence, all web services are API but not vice versa.



RESTful Web Services

What is REST?

REST (Representational State Transfer) is a set of principles for designing networked applications. RESTful web services use these principles to create APIs (Application Programming Interfaces) that allow different systems to communicate over the internet.

Key Ideas Behind RESTful Web Services:

1. **Resources:** Imagine everything as a resource, like pages in a book. In a shopping website, resources could be products, orders, and customers.
2. **Standard Methods:** Just like how we have standard ways to open a door (push or pull), RESTful web services use standard methods for interacting with resources. These methods include:
 - a. *GET*: To read or fetch information.
 - b. *POST*: To create something new.
 - c. *PUT*: To update something.
 - d. *DELETE*: To remove something.
3. **Statelessness:** Each time you ask for something, you provide all the information needed. Imagine every time you visit a library, you bring all the details about the book you want to borrow. The library doesn't need to remember you between visits.
4. **Representation:** Resources can be represented in different formats like JSON (easy for computers and humans to read) or XML (more structured format).

How RESTful Web Services Work in a Restaurant Example:

1. *Viewing the Menu*

- HTTP Method: GET
- Endpoint: /menu
- Action: When a customer wants to view the menu, they make a GET request to the /menu endpoint. The server responds with the menu details in a format like JSON.

2. *Placing an Order*

- HTTP Method: POST
- Endpoint: /order
- Action: When a customer places an order, they send a POST request to the /order endpoint with the order details. The server processes the order and responds with a confirmation.

3. *Checking Order Status*

- HTTP Method: GET
- Endpoint: /order/{orderId}/status
- Action: To check the status of an order, the customer makes a GET request to the /order/{orderId}/status endpoint, where {orderId} is the ID of the order. The server responds with the current status of the order.

4. *Getting Restaurant Information*

- HTTP Method: GET
- Endpoint: /info
- Action: When a customer wants to get information about the restaurant, they make a GET request to the /info endpoint. The server responds with details like the address and opening hours.

Benefits of RESTful Web Services:

- **Simplicity:** Easy to use and understand, leveraging standard HTTP methods.
- **Scalability:** Stateless nature allows servers to handle multiple requests efficiently.
- **Flexibility:** Resources can be accessed and manipulated using a standard set of operations, making it easier to integrate with various clients (web browsers, mobile apps).
- **Interoperability:** Works well with other web standards, making it easy to integrate with other systems.

Conclusion

RESTful web services are a powerful yet straightforward way for applications to communicate over the internet. They enable apps and websites to exchange information and perform actions using standard methods. This makes it easier for developers to build flexible, efficient, and interoperable systems that can work together seamlessly.

References:

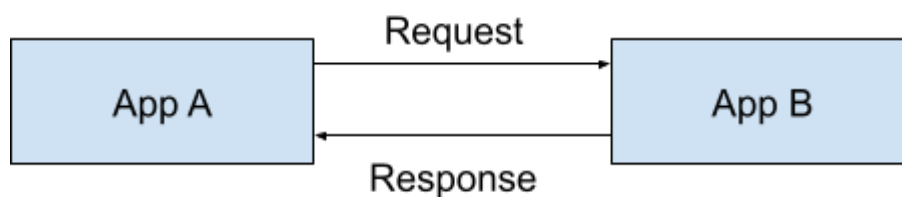
- [What is the difference between API and REST API](#)
- [What are protocols?](#)
- [Request vs Response](#)
- [JSON vs XML](#)
- [REST vs SOAP](#)
- [Popular Interview question on Web Services](#)

Rest API and Spring Boot

Introduction to Web Services

Web services are a way to interact with different software systems over the Internet. They allow us to load and send data between systems, enabling integration and communication between different software applications. Web services use standardized protocols and interfaces like HTTP and XML to exchange information and perform tasks. This makes it possible for different software systems to communicate and exchange data, regardless of the specific technologies or platforms they are built on. Web services are an important part of modern software development and are widely used in various applications, including e-commerce, data integration, and business automation.

Data exchange between applications



Assume there is an application A, which needs some data from another application B. Application A will send a request to application B with the appropriate requirement using some standardized protocol. Application B would process the request and send the response back to application A. The application can then use this data according to its needs.

There are 2 most popular ways to exchange data between applications - XML and JSON.

Let's assume that we want to share the following details of a Student -

Student [ID - 123, Name - Tester]

XML

XML is the abbreviation for Extensible Markup Language. In XML, we would format the data as follows -

```
<Student>
  <ID> 123 </ID>
  <Name> Tester </Name>
</Student>
```

JSON

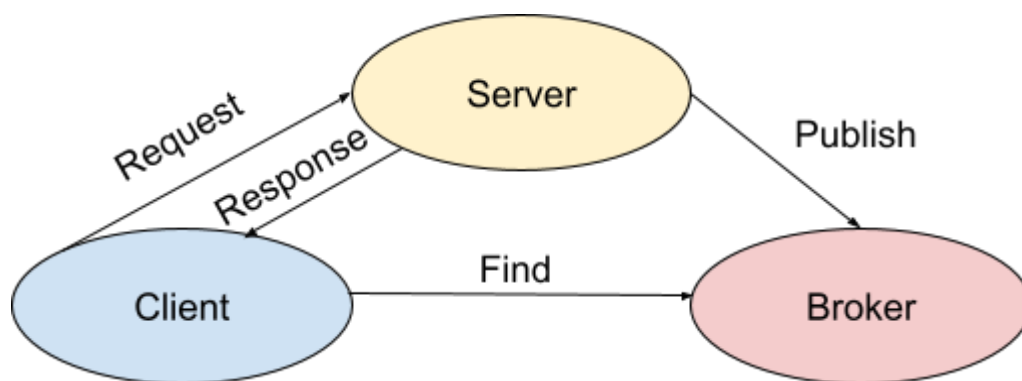
Javascript Object Notation, also known as JSON, is the most popular data exchange format. In JSON, we would format the data as follows -

```
{ "Student" :  
  [ { "ID" : 123,  
      "Name" : "Tester"  
    } ]  
}
```

Key Terminologies

- **Request and Response** - The input which comes to the web service is known as a Request, and the output given by the web service is known as a Response.
- **Message exchange format** - It is the format in which data exchange takes place. There are two most popular exchange formats - XML and JSON.
- **Service Definition** - It ensures that the process of data transfer is standardized.
- **Service Consumer or Client** - The application which sends the request is known as the service consumer or client.
- **Service Provider or Server** - The application which processes the request and gives back the Response is known as Service Provider or Server.
- **Transport** - It is responsible for the data exchange between the service provider and consumer. The data transfer can take place via HTTP or Queues.

Web Services Architecture



1. As soon as the server is available, it will publish a service description to the Broker.
2. Then the Service consumer or client can retrieve the service description from the Broker.
3. Finally, the client can send the Request to the Server and the Server can send the response.

All web services are APIs, but not all APIs are web services.

Introduction to RESTful Web Services

RESTful web services are loosely coupled, lightweight web services that are particularly well suited for creating APIs for clients spread out across the internet.

RESTful Key Elements

- **Resources** - In RESTful web services, a resource is a specific piece of information or data that can be accessed and manipulated using standard HTTP methods (such as GET, POST, PUT, and DELETE). The information or data is typically represented in a format such as JSON or XML and is identified by a unique URI (Uniform Resource Identifier).
- **Request Headers** - The header is an additional piece of information provided with the request. It contains authorization data or the type of response required by the client.
- **Request Body** - The data which is sent by the client in the request to be processed by the server is known as Request Body.
- **Response Body** - If the client asks for some piece of information, the data is sent in the response body by the server.
- **Response Status Code** - The server makes use of various status codes to notify if the request was successful or if not, then the type of error encountered.
- **Request Methods** - Request methods tell about the type of request to the server.



- GET method is used to fetch data from the server.
- POST method is used to store new data on the server.
- PUT method is used to update existing data in the server.
- DELETE method is used to delete the data which exists on the server.

SOAP vs REST web services

SOAP Protocol	RESTful Web Services
SOAP is a protocol.	REST is an architectural approach.
The data exchange format is always XML.	There is no strict data exchange format.
SOAP cannot use RESTful services because it is a protocol.	RESTful services can use SOAP web services because it is an architectural approach that can use any protocol like HTTP or SOAP.

Cloud-based architecture work on the REST principle, it makes more sense for web services to be programmed on the REST services based.

RESTful Web Services with Spring Boot

REST API - Important Annotations

- **@RestController** - It is a Spring annotation used to create a RESTful web service using Spring Boot. It is used to create web services that return JSON or XML data instead of HTML views. When a request is made to a URL that is mapped to a method annotated with **@RestController**, the method is invoked and the return value is automatically converted to JSON or XML and sent back to the client as the HTTP response.
- **@RequestMapping** - **@RequestMapping** is a Spring annotation that is used to map a web request to a specific method in a controller class.
 - It is used to handle requests to a specific URL and can be applied at the class level or the method level.
 - The **@RequestMapping** annotation can also be used to specify the HTTP method (such as GET, POST, PUT, DELETE) that the mapped method should handle, and can be used to map requests to specific HTTP headers or request parameters.
- **@GetMapping** - **@GetMapping** is a Spring MVC annotation that is used to handle HTTP GET requests. It maps a specific URI (Uniform Resource Identifier) pattern to a controller method, allowing it to handle the incoming request and return a response.

Creation of Hello World REST API

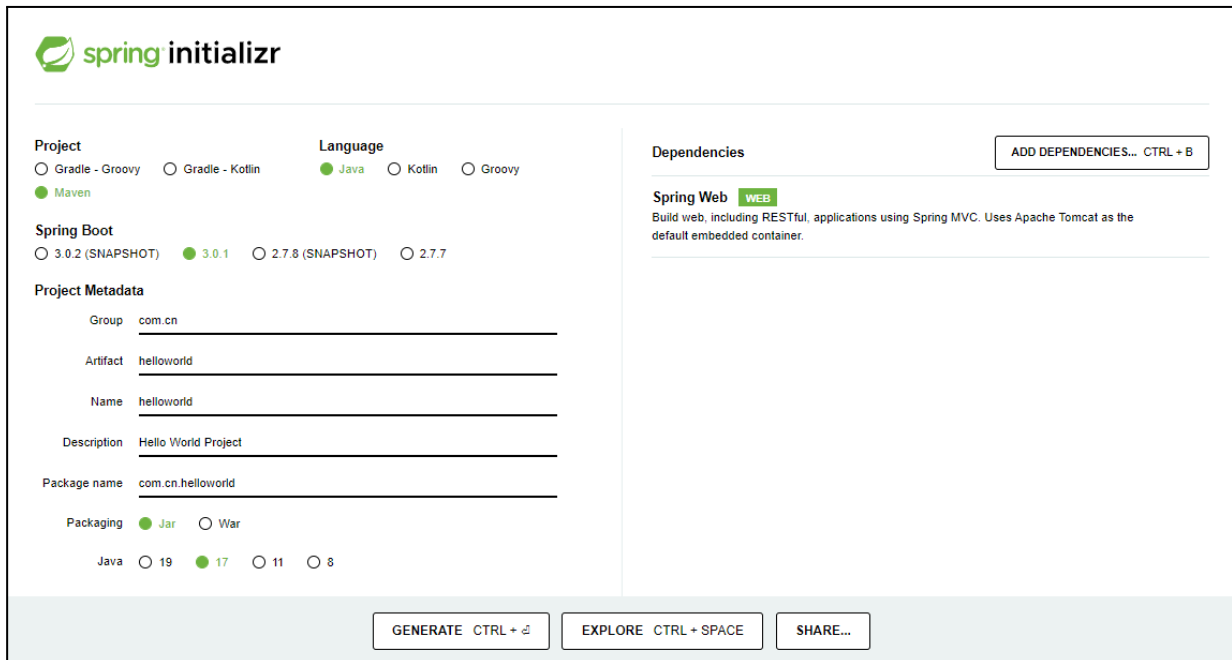
We would be creating a Maven project for the Hello World REST API. Thus, it is required that we have the following tools and technologies for our project.

- Spring Boot version 2.2.2+
- Java 8+
- Maven 3.0+
- An IDE of your choice (The tutorial uses Eclipse IDE)
- Dependency - Spring Web

@RestController, @RequestMapping and @GetMapping

1. Visit <https://start.spring.io> (This is a web-based Spring Initializer).

Add Spring Web Dependency to your project.



The screenshot shows the Spring Initializr web interface. The 'Project' section has 'Maven' selected. The 'Language' section has 'Java' selected. The 'Spring Boot' section has '3.0.1' selected. The 'Project Metadata' section has the following values: Group: com.cn, Artifact: helloworld, Name: helloworld, Description: Hello World Project, Package name: com.cn.helloworld, Packaging: Jar, Java: 17. The 'Dependencies' section has 'Spring Web' selected. At the bottom, there are buttons for 'GENERATE', 'EXPLORE', and 'SHARE...'. The 'GENERATE' button has a tooltip that says 'CTRL + G'.

In the end, the selections should look like this, and then click on **GENERATE** button on the bottom. It would download a zip file.

2. Extract the zip in a folder.
3. Import the project in Eclipse by going to -
File -> Open projects from File System -> Directory -> Select the extracted folder -> Finish.
4. Create a new Controller class in the project as follows.

```
@RestController
@RequestMapping("/project")
public class HelloWorldController {
    @RequestMapping("/hello")
    public String hello()
    {
        return "Hello Coding Ninjas !";
    }
}
```

```
}

@GetMapping("/hello-world")
public String helloWorld()
{
    return "Hello from Earth !";
}
}
```

5. Run the project and go to your browser and visit the URL - <http://localhost:8080/project/hello-world> to see the output.
6. In this example, we made use of `@RestController` on the class which helps us to return JSON from the methods, `@RequestMapping` to set a base URL for the class, and then `@GetMapping` to handle get requests on a URL.

@PathVariable

Now, let's add another method to our controller class, which can grab a value from the URL dynamically. This can be done with the help of `@PathVariable` annotation in the method parameters.

```
@RestController
@RequestMapping("/project")
public class HelloWorldController {

    @RequestMapping("/hello")
    public String hello()
    {
        return "Hello Coding Ninjas !";
    }

    @GetMapping("/hello-world")
    public String helloWorld()
    {
        return "Hello from Earth !";
    }

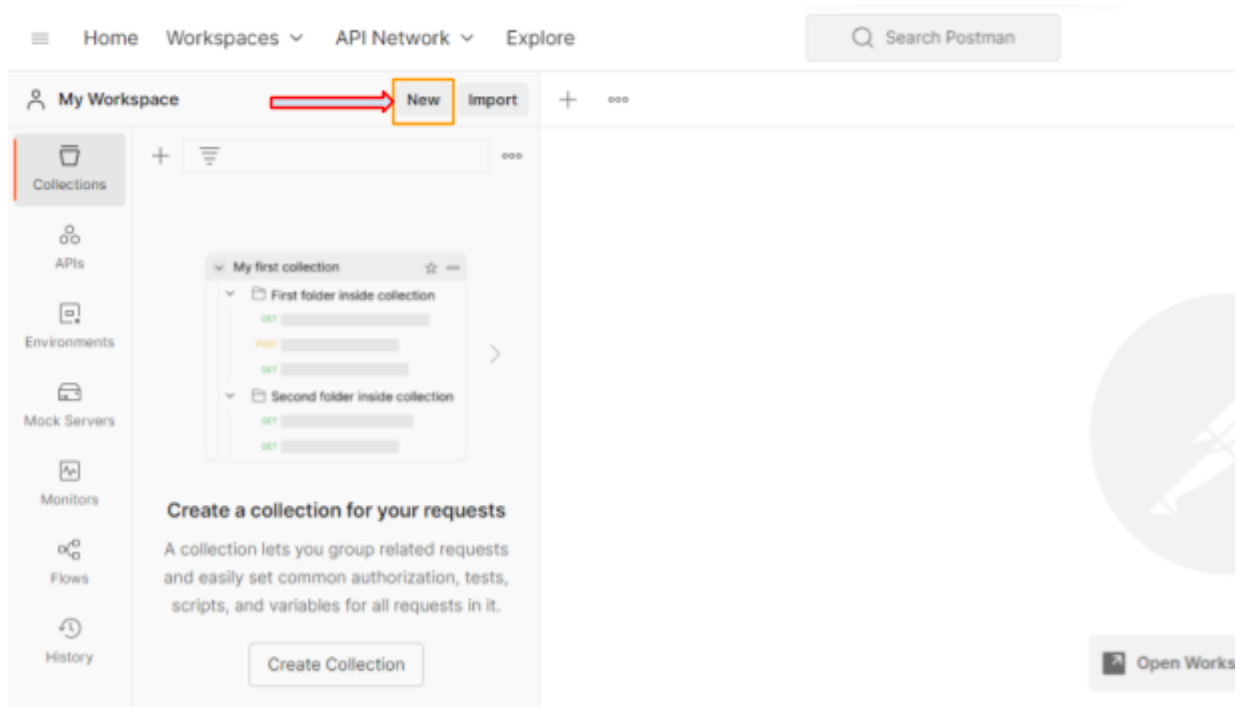
    @GetMapping("/hello-mars/{name}")
    public String helloFromMars(@PathVariable String name)
    {
        return "Hello from Mars "+name;
    }
}
```

Now, the controller class would look something like this. Here `@PathVariable` takes the `{name}` from the URL and attaches it to the 'name' parameter of the 'helloFromMars' method. This parameter can then be used in the method body normally.

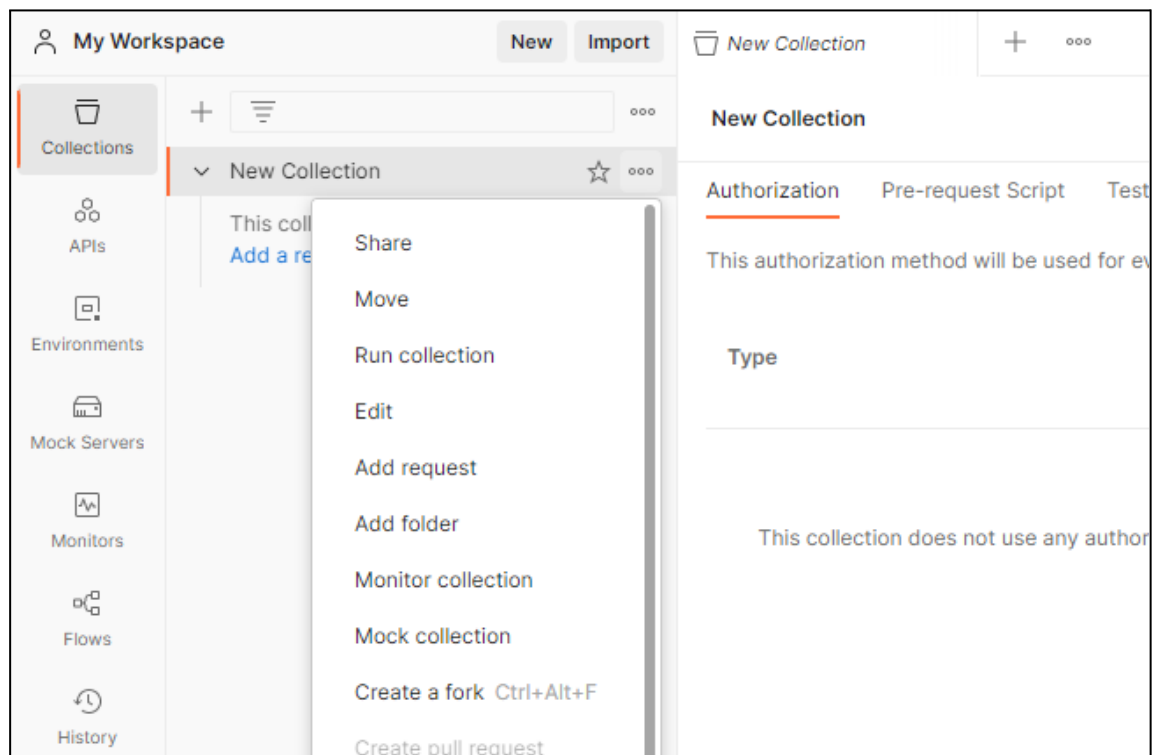
Now visit the URL <http://localhost:8080/hello-mars/Elon> , you would see an output that says - “Hello from Mars Elon”.

Post Man Setup

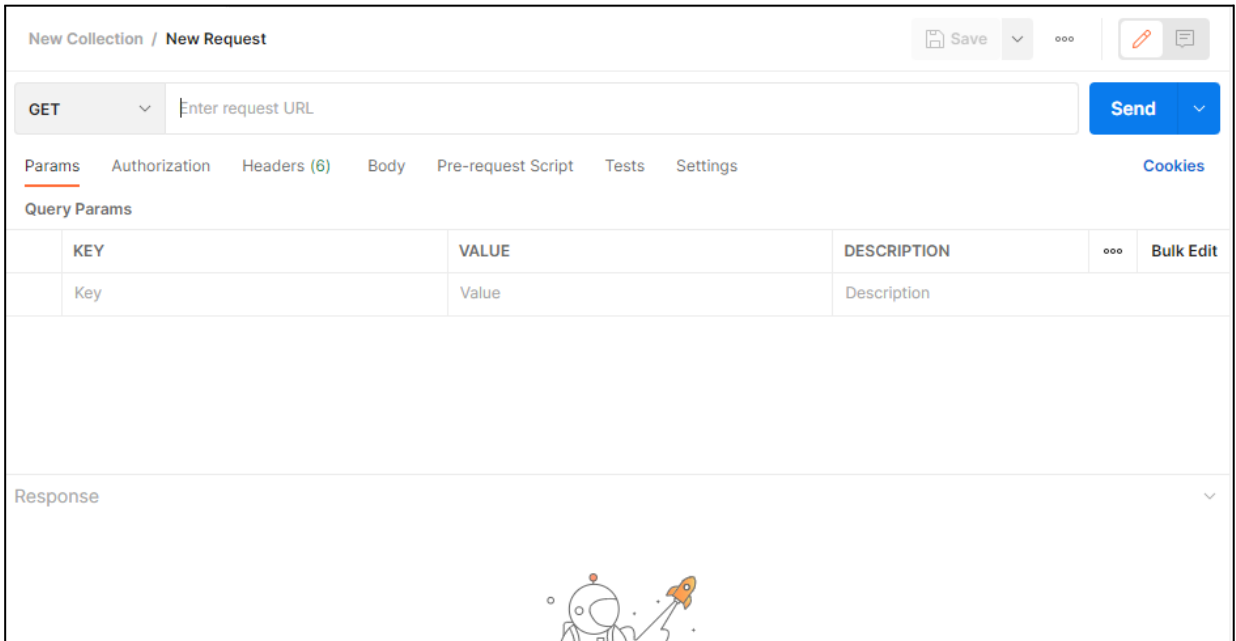
1. Visit <https://www.postman.com/downloads/> and download the PostMan app.
2. Install the Postman app and open it. You would see a screen like this.



3. Click on New, and select “Collection” from the screen.
4. Right-click on “New Collection” and select ‘New Request’ from the menu.



- Now you have added a request, and you are ready to test APIs.



New Collection / New Request

GET Enter request URL Send

Params Authorization Headers (6) Body Pre-request Script Tests Settings Cookies

Query Params

KEY	VALUE	DESCRIPTION	...	Bulk Edit
Key	Value	Description		

Response

- You can enter the URL to be tested, and also select from a range of request methods such as GET, POST, PUT, DELETE etc and see the response in the bottom section.

Implementing the POST method

- Create a new Project from the web [spring.initialzr](https://spring.io/guides-topics/docs/spring-initializr/).
- Create a Hotel class as follows

```
public class Hotel {
    private String id;
    private String name;
    private long rating;
    private String city;

    public Hotel(String id, String name, long rating, String city) {
        super();
        this.id = id;
        this.name = name;
        this.rating = rating;
        this.city = city;
    }

    public String getId() {
        return id;
    }

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

    public String getName() {
```

```
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public long getRating() {
        return rating;
    }
    public void setRating(long rating) {
        this.rating = rating;
    }
    public String getCity() {
        return city;
    }
    public void setCity(String city) {
        this.city = city;
    }
}
```

3. Create a HotelService class which will contain business logic, and methods to add, get or delete Hotels, which will be called by Controller class.

We will make use of ArrayList of Hotel to store the hotels, and also map the Hotels to their ID's in a HashMap, so that they can be fetched easily later on.

```
@Service
public class HotelService {

    List<Hotel> hotelList = new ArrayList<>();
    Map<String,Hotel> hotelMap= new HashMap<>();

    public void createHotel(Hotel hotel) {
        hotelList.add(hotel);
        hotelMap.put(hotel.getId(), hotel);
    }
}
```

4. Create a new Controller class and annotate it as @RestController.

```
@RestController
@RequestMapping("/hotel")
public class HotelController {
    @Autowired
    HotelService hotelService;

    @PostMapping("/create")
    public void createHotel(@RequestBody Hotel hotel)
```

```
    {  
        hotelService.createHotel(hotel);  
    }  
}
```

Implementing the GET method

In the Hotel project's service class, let's add a service method which will return the list of all Hotels, and a service method which returns a Hotel by its ID.

So, in the Service class add two methods as follows:

```
@Service  
public class HotelService {  
  
    List<Hotel> hotelList = new ArrayList<>();  
    Map<String,Hotel> hotelMap= new HashMap<>();  
  
    public Hotel getHotelById(String id) {  
        Hotel hotel = hotelMap.get(id);  
        return hotel;  
    }  
  
    public List<Hotel> getAllHotels() {  
        return hotelList;  
    }  
  
    public void createHotel(Hotel hotel) {  
        hotelList.add(hotel);  
        hotelMap.put(hotel.getId(), hotel);  
    }  
}
```

Also add respective controller methods in the Controller class with the help of `@PathVariable` to get the HotelId from the URL.

```
@RestController  
@RequestMapping("/hotel")  
public class HotelController {  
    @Autowired  
    HotelService hotelService;  
  
    @GetMapping("/id/{id}")  
    public Hotel getHotelById(@PathVariable String id)  
    {  
        return hotelService.getHotelById(id);  
    }  
  
    @GetMapping("/getAll")
```

```
public List<Hotel> getAllHotels()
{
    return hotelService.getAllHotels();
}

@PostMapping("/create")
public void createHotel(@RequestBody Hotel hotel)
{
    hotelService.createHotel(hotel);
}
}
```

Implementing the DELETE Method

Now let's add a method to delete the Hotel by a given ID, so add a method in the service class as follows:

```
@Service
public class HotelService {

    List<Hotel> hotelList = new ArrayList<>();
    Map<String,Hotel> hotelMap= new HashMap<>();

    public void deleteHotelById(String id) {
        Hotel hotel = getHotelById(id);
        hotelList.remove(hotel);
        hotelMap.remove(id);
    }

    public Hotel getHotelById(String id) {
        Hotel hotel = hotelMap.get(id);
        return hotel;
    }

    public List<Hotel> getAllHotels() {
        return hotelList;
    }

    public void createHotel(Hotel hotel) {
        hotelList.add(hotel);
        hotelMap.put(hotel.getId(), hotel);
    }
}
```


And similarly in the Controller class, make use of `@DeleteMapping` and `@PathVariable` to map a URL to delete a Hotel by Id as follows :

```
@RestController
@RequestMapping("/hotel")
public class HotelController {
    @Autowired
    HotelService hotelService;

    @DeleteMapping("/remove/id/{id}")
    public void deleteHotelById(@PathVariable String id)
    {
        hotelService.deleteHotelById(id);
    }

    @GetMapping("/id/{id}")
    public Hotel getHotelById(@PathVariable String id)
    {
        return hotelService.getHotelById(id);
    }

    @GetMapping("/getAll")
    public List<Hotel> getAllHotels()
    {
        return hotelService.getAllHotels();
    }

    @PostMapping("/create")
    public void createHotel(@RequestBody Hotel hotel)
    {
        hotelService.createHotel(hotel);
    }
}
```

Implementing the PUT Method

Let's add a method in the service class which takes a new Hotel object and replaces the old Hotel object with the same ID.

So, add a new update method in the existing service class as follows:

```
@Service
public class HotelService {

    List<Hotel> hotelList = new ArrayList<>();
    Map<String,Hotel> hotelMap= new HashMap<>();

    public void updateHotel(Hotel updatedHotel) {
```

```
//1. Get the previous data of the hotel
//2. remove this old data from list
//3. Add the updated data to the list.

Hotel existingHotel= getHotelById(updatedHotel.getId());
hotelList.remove(existingHotel);
hotelList.add(updatedHotel);

//4. update the previous data with new data.
//5. Update the map with new data.

hotelMap.put(updatedHotel.getId(), updatedHotel);
}

public void deleteHotelById(String id) {
    Hotel hotel = getHotelById(id);
    hotelList.remove(hotel);
    hotelMap.remove(id);
}

    public Hotel getHotelById(String id) {
        Hotel hotel = hotelMap.get(id);
        return hotel;
    }

    public List<Hotel> getAllHotels() {
        return hotelList;
    }

    public void createHotel(Hotel hotel) {
        hotelList.add(hotel);
        hotelMap.put(hotel.getId(), hotel);
    }
}
```

Accordingly add a update method in the controller class with the help of `@PutMapping` as follows :

```
@RestController
@RequestMapping("/hotel")
public class HotelController {
    @Autowired
    HotelService hotelService;

    @PutMapping("/update")
    public void updateHotel(@RequestBody Hotel hotel)
    {
```

```
        hotelService.updateHotel(hotel);
    }

    @DeleteMapping("/remove/id/{id}")
    public void deleteHotelById(@PathVariable String id)
    {
        hotelService.deleteHotelById(id);
    }

    @GetMapping("/id/{id}")
    public Hotel getHotelById(@PathVariable String id)
    {
        return hotelService.getHotelById(id);
    }

    @GetMapping("/getAll")
    public List<Hotel> getAllHotels()
    {
        return hotelService.getAllHotels();
    }

    @PostMapping("/create")
    public void createHotel(@RequestBody Hotel hotel)
    {
        hotelService.createHotel(hotel);
    }
}
```

Exception Handling

Let's say in our Hotel Application, we have a list of Hotels with ID's 1,2,3,4 and 5 respectively. And a user sends a get request with an ID of 7 which does not exist in the list. So, we should be able to handle such requests and also respond with proper error messages in such cases. This is known as exception handling, and there are several HTTP response status codes which indicate whether a given request was successful or not.

These are some of the most common response codes -

- **404** - This indicates that the requested resource was not found on the server.
- **400** - It is used to indicate that the server did not process the request due to an error in the request sent by the user.
- **500** - This status code indicates that the server encountered an internal error and could not process the request.

In Spring Boot we can annotate an exception class with `@ResponseStatus`, so if that exception is thrown, then a response status is sent to the client.

Implementing Exception Handling

We will start by creating a new class which extends the `RuntimeException` class and also annotate it with `@ResponseStatus`. In the `@ResponseStatus`, pass the `HttpStatus.NOT_FOUND`, as the argument. There are many options which are available to us, such as `HttpStatus.OK`, `HttpStatus.BAD_REQUEST` etc. Then add a constructor which takes a `String` as an input and passes it to the super class constructor.

So, your new Class would look like this -

```
@ResponseStatus(HttpStatus.NOT_FOUND)
public class HotelNotFoundException extends RuntimeException{

    public HotelNotFoundException(String message) {
        super(message);
    }
}
```

Our custom exception class is ready, now we can throw this exception in our service method. So, in the `getHotelById` method, add a condition which will check if a Hotel exists by the given id. We can do this with the help of `isEmpty` method of `ObjectUtils` class. It is a static method which takes an object as argument and returns true if it is null. The `getHotelById` method should look like this now:

```
public Hotel getHotelById(String id) {
    if(ObjectUtils.isEmpty(hotelMap.get(id)))
    {
        throw new HotelNotFoundException("Hotel not found for
id: "+id);
    }
    Hotel hotel = hotelMap.get(id);
    return hotel;
}
```

Validations for RESTful Services

Validating the user input is a common requirement, and thus Java provides with several annotations to help constraint the Entity according to various criterias.

Some of the most common annotations are:

- `@Size` validates that the property has a size or length as specified by the min and max attributes. It can be applied to Collections, Map, Strings, and arrays.
- `@Min` is used to validate that the value of field should not be lesser than that given in annotation attribute.
- `@Max` validates that the annotated property has a value not greater than the value attribute.

We will make use of these annotations to restrict the Hotel field values as follows:

```
private String id;

@Size(min=3)
private String name;

@Min(value = 1)
@Max(value = 10)
private long rating;
private String city;
```

After these restrictions, the 'name' will have a minimum length of 3, rating will have a minimum value of 1 and maximum value of 10.

These annotations only provide a restrictions on the fields, but the validation is taken care by another annotation which is `@Valid`. Let's have a look at the implementation of `@Valid`.

@Valid

`@Valid` can be used against a method parameter in the controller method which will accept the input from the user. So, in our hotel project we will use the `@Valid` in our `createHotel` controller method. So, after using `@Valid`, our controller method would look like this -

```
@PostMapping("/create")
public void createHotel(@Valid @RequestBody Hotel hotel)
{
    hotelService.createHotel(hotel);
}
```

BindingResult Class

`@Valid` will validate our inputs, but we do need to throw an exception if the input is not valid. So we make use of `BindingResult` class to store the result or the errors that may have occurred. This can be passed as a parameter to the controller method. Then we can make use of `hasErrors` method of `BindingResult` class to detect if any error has occurred, while validating the input.

So after passing `BindingResult` class as a parameter and adding a if condition to check errors and throw exception, our `createHotel` method would be as follows:

```
@PostMapping("/create")
public void createHotel(@Valid @RequestBody Hotel hotel,
BindingResult
bindingResult)
{
    if(bindingResult.hasErrors())
    {
        throw new RuntimeException("Request Not Valid");
    }
}
```

```
        hotelService.createHotel(hotel);  
    }
```

Instructor Codes

- [Hotel Application](#)

Introduction to Rest TEMPLATE

What is a Rest Template?

We can have different services in a single application and use Rest Templates to provide **intercommunication** between these services. The Rest Template is the central class within the Spring framework for executing synchronous HTTP requests on the client side.

Use of Rest Template

- Creating a URL object and opening the connection
- Configuring the HTTP request
- Executing the HTTP request
- Interpretation of the HTTP response
- Converting the HTTP response into a Java object
- Exception handling

Example:

Let's say our application has two service module and their respective models:

- **Hotel Module:** Provides services for creating, adding, deleting and updating hotels and their ratings.
- **Rating Module:** Provides create, add, delete and update services for key, value pair of <HotelId, Hotel Rating>.

Task: Let's say we have a service that returns a hotel by id, and we want an updated rating from the Rating Module.

Solution: With RestTemplate, we create a service that calls the Rating Module's service for updated rating.

```
//HotelService.java
public Hotel getHotelById(String id) {
    if (ObjectUtils.isEmpty(hotelMap.get(id))) {
        throw new HotelNotFoundException("Hotel not found for id: " + id);
    }
    Hotel hotel = hotelMap.get(id);
    //rest service to fetch the rating by id
    return hotel;
}
```

When using RestTemplate, all these things happen in the background, and the developer doesn't have to bother with it. Rest Template makes it convenient for services to communicate with each other.

Rest Template Builder

RestTemplate Builder helps us to create a RestTemplateBuilder Object. The rest template object is created with the help of RestTemplateBuilder (a class), which contains methods of the rest template. It also provides convenient methods to register converters, error handlers, and UriTemplateHandlers.

Example:

This template is created in the Hotel module, and this service is used to call rating services in the Rating Module.

```
//RatingServiceCommunicator.java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.web.client.RestTemplateBuilder;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;
@Service
public class RatingServiceCommunicator {

    private final RestTemplate restTemplate;

    @Autowired
    public RatingServiceCommunicator (RestTemplateBuilder
restTemplateBuilder) {
        this.restTemplate=restTemplateBuilder.build();
    }
}
```

In the code above; the RatingServiceCommunicator class uses RestTemplate to communicate with the rating service in the Rating module. The RestTemplateBuilder is autowired into the class constructor, allowing Spring to provide an instance of RestTemplateBuilder.

The restTemplateBuilder.build() method is called to create a RestTemplate object. This method creates a new instance of RestTemplate with any customisations applied using the RestTemplateBuilder instance.

Once the RestTemplate object is created, it can be used within the RatingServiceCommunicator class to make HTTP requests to the rating service.

Valuable Methods for Rest Template

Methods primarily used in Rest Template are:

- `getForEntity()`
- `getForObject()`
- `postForObject()`
- `exchange()` - Common method for all GET, PUT, POST, and Delete operations.

For additional information on the methods of RestTemplate, please refer to the [Javadoc](#).

Method 1: `getForEntity()`

The `getForEntity()` method retrieves resources from the HTTP Request. We have to pass the URL and response type, and it returns the response as `ResponseEntity`, using which we can get the response status code, response body, and much more helpful information. We can also add other variables to request if required.

Example:

We will use the rest template object created above for the `getForEntity` method.

Two parameters in the request:

- URL of the HTTP Request
- Return Type - which will store the response in the type mentioned.

```
//RatingServiceCommunicator.java
public Long getRating (String id){
    String url="http://localhost:8081/rating/id/";
    ResponseEntity<Long> response = restTemplate.getForEntity(url+id,
    Long.class);
    return response.getBody();
}
```

This response gets stored in `ResponseEntity` with a mentioned type (e.g., `Long`).

`ResponseEntity` encapsulates the HTTP response status code, the HTTP headers, and the body that has already been converted into a Java object. It will give us methods like `getStatusCode()`, `getHeaders()`, and `getBody()`.

Method 2: getForObject()

This method is similar to the `getForEntity()` method, as it requires the URL and the return Type in parameters, the only difference is, it will directly return only the value of the response body, so it will not contain methods like `getHeaders()` and `getResponseBody()`.

Example:

If you only require the response value, use the `getForObject()` method.

```
RatingServiceCommunicator.java
public long getRating (String id)
String url="http://localhost:8081/rating/id/";
//ResponseEntity<Long> response= restTemplate.getForEntity(url+id,
Long.class);
Long ratingResponse = restTemplate.getForObject(url+id, Long.class);
return ratingResponse;
}
```

Method 3: postForObject()

This method creates a new resource using the HTTP POST method. It takes the URL, request value, and response type as input and returns an entity of the object created.

Example:

Task: Let's say we add a new hotel, and we also want to call a method in the "Rating Module" to add a rating for this new hotel.

Solution: The Hotel Module will use `postForObject()` in its service to call the service in the image below, which is present in the Rating Module. This service adds a `<HotelId, RatingValue>` pair into the Rating Model.

```
RatingService.java
@PostMapping(path = "/add")
public void addRating(@RequestBody Map<String, Double>
hotelRatingMap) {
    ratingService.addRating(hotelRatingMap);
}
```

Simply, we will create an `addRating` method in hotel service, which will take `<HotelId, RatingValue>` map and pass it as a parameter `postForObject`, which will save `Object.class` as a response, as we are not returning anything.

```
RatingServiceCommunicator.java
public void addRating (Map<String, Long> ratingsMap) {

String url="http://localhost:8081/rating/add";
restTemplate.postForObject(url, ratingsMap, Object.class);

}
```

Create hotel Service will call our “addRating” service in RatingServiceCommunicator.java.

```
HotelService.java
public void createHotel (Hotel hotel) {

Map<String, Long> ratingsMap = new HashMap<>();

hotelList.add(hotel); hotelMap.put(hotel.getId(), hotel);
ratingsMap.put(hotel.getId(), hotel.getRating());
ratingServiceCommunicator.addRating(ratingsMap);
}
```

Method 4: exchange()

Exchange is the method through which we can call all HTTP requests, so we can use this method in place of calling specific HTTP methods like getForObject(), postForObject(), etc. The exchange method can be used for HTTP DELETE, GET, HEAD, OPTIONS, PATCH, POST, and PUT methods.

The exchange is a method to fetch, update, create new and delete older data from our application. The exchange method returns ResponseEntity, using which we can get the response status, body, and headers.

HTTP Entity

HttpEntity is a parameter we pass in the exchange method.

- It contains your request, wrapped in an HTTP entity. It will contain both headers and a request body.
- We can create this entity by passing both body and header in the input.

Parameters of exchange() method

- URL
- HTTP Request Method
- Request Entity
- Response Type
- Other URI Variables

- `ResponseEntity` is the return type of this method.

```
ResponseEntity<T> exchange(String url, HttpMethod method, HttpEntity<?>
requestEntity, Class<T> responseType, Map<String,?> uriVariables)
```

```
RatingServiceCommunicator.java
public void addRating(Map<String, Long> ratingsMap) {
    String url = "http://localhost:8081/rating/add";
    //restTemplate.postForObject(url, ratingsMap, Object.class);
    HttpEntity<Map<String, Long>> requestEntity = new
HttpEntity<>(ratingsMap);

    restTemplate.exchange(url,HttpMethod.POST,requestEntity,Object.class);
}
```

exchange(): PUT

To use the `exchange()` to update data, we need to use the HTTP method as **`HttpMethod.PUT`**. For request entities, we can use `HttpEntity`.

Example:

Task: If we update our rating in hotel service, we also want the ratings to be updated in the Rating module.

Solution: A put request is used in the rating service of the Hotel Module to communicate with the Rating module.

```
RatingService.java
@PutMapping (path = "/update")
public void updateRating(@RequestBody Map<String, Double> hotelRatingMap) {
    ratingService.updateRating(hotelRatingMap);
}
```

This method will call the `updateRating` method in the Rating Module. For this, we will use the **`HttpMethod.PUT`** request.

```
RatingServiceCommunicator.java
public void updateRating(Map<String, Long> ratingsMap)
{
    String url = "http://localhost:8081/rating/update";

    HttpEntity<Map<String, Long>> requestEntity = new
HttpEntity<>(ratingsMap);
    restTemplate.exchange(url,HttpMethod.PUT,requestEntity,Object.class);
}
```

UpdateHotel will call the above method to update the rating.

```
HotelService.java
public void updateHotel(Hotel updatedHotel) {
    //1. Get the previous data of the hotel
    //2. remove this old data from list
    //3. Add the updated data to the list.

    Hotel existingHotel= getHotelById(updatedHotel.getId());

    hotelList.remove(existingHotel);
    hotelList.add(updatedHotel);

    //4. update the previous data with new data.
    //5. Update the map with new data.

    hotelMap.put(updatedHotel.getId(), updatedHotel);

    Map<String,Long> updatedRating = new HashMap<>();
    updatedRating.put(updatedHotel.getId(), updatedHotel.getRating());
    ratingServiceCommunicator.updateRating(updatedRating);
}
```

exchange(): DELETE

To use the exchange() to delete data, we need to use the HTTP method as **HttpMethod.DELETE**. No request entity is required for this method, so it will pass null for the same.

```
RatingService.java
@DeleteMapping(path = "/id/{id}")
public Double getRatingById(@PathVariable String id) {
    return ratingService.deleteRating(id);
}
```

```
RatingServiceCommunicator.java
public void deleteRating(String id)
{
    String url ="http://localhost:8081/rating/remove/id/";
    restTemplate.exchange(url+id,HttpMethod.DELETE,null,Object.class);
}
```

```
HotelService.java
public void deleteHotelById(String id) {
    Hotel hotel = getHotelById(id);
    hotelList.remove(hotel);
    hotelMap.remove(id);
    ratingServiceCommunicator.deleteRating(id);
}
```

By using the `exchange()` method, you can handle different types of HTTP requests (DELETE, GET, HEAD, OPTIONS, PATCH, POST, and PUT) in a flexible manner. It lets you specify the HTTP method, request entity (body and headers), response type, and other parameters.

In the provided examples, the response type is set to `Object.class`, which means the expected response can be of any type. However, you can modify the `Object.class` to match the specific type of response you expect. For example, if you expect the response to be a `String`, you can set the response type as `String.class`.

Overall, the `exchange()` method in `RestTemplate` provides versatility and customisation options for making HTTP requests in your application, enabling you to handle various scenarios and interact with different APIs effectively.

HTTP Error Handling

Error handling is the process comprised of anticipation, detection, and resolution of application errors. The tasks of the Error Handling process are to detect each error, report it to the user, and then make some recovery strategies and implement them to handle the error. We will use try-and-catch blocks to handle exceptions in our code.

When the hotel service communicates with the rating service, there are changes or errors, like the rating service getting modified or the URL passed in the hotel service being incorrect. So, we have to handle these kinds of errors.

By default, the RestTemplate will throw one of these exceptions in the case of an HTTP error:

- **HttpClientErrorException:** Client errors occur when the client side of a communication process encounters issues while interacting with a server. These errors are typically indicated by HTTP status codes in the **4xx** range. Here are some common client error status codes:
 - **400 Bad Request:** This status code indicates that the server could not understand the client's request due to malformed syntax or invalid parameters. It often occurs when the client sends a request the server cannot process.
 - **401 Unauthorized:** When a client receives a 401 status code, the requested resource requires authentication. The client must provide valid credentials (e.g., username and password) to access the resource.
 - **403 Forbidden:** The server understands the client's request, but the client cannot access the requested resource. This status code differs from 401, implying the client lacks the necessary permissions to access the resource even after authentication.
 - **404 Not Found:** This status code indicates that the server could not find the requested resource. It commonly occurs when a client tries to access a URL or endpoint that doesn't exist on the server.
- **HttpServerErrorException:** Server errors occur when there are issues on the server side of a communication process. These errors are typically indicated by HTTP status codes in the **5xx** range. Here are some common server error status codes:
 - **500 Internal Server Error:** This is a generic error message indicating that an unexpected condition occurred on the server, preventing it from fulfilling the client's request. It signifies an error within the server's infrastructure or application code.
 - **504 Gateway Timeout:** This status code indicates that the server, acting as a gateway or proxy, did not receive a timely response from an upstream server to complete the request. It often occurs when there is a delay or timeout between different servers.
- **UnknownHttpStatusCodesException:** The `UnknownHttpStatusCodesException` is an exception by RestTemplate when it encounters an unrecognized or unknown HTTP status code during communication with a server. This exception provides a way to

handle scenarios where the server returns custom or non-standard status codes. By catching the `UnknownHttpStatusCodeException`, developers can implement specific error-handling logic and access details about the unknown status code, such as the numeric value and associated message. This enables better control and informed handling of unknown status codes in the application.

All of these exceptions are extensions of ***RestClientResponseException***.

```
RatingServiceCommunicator.java
public void deleteRating(String id)
{
    String url = "http://localhost:8081/rating/remove/id/";
    try {
        restTemplate.exchange(url+id, HttpMethod.DELETE, null, Object.class);
    }
    catch (HttpClientErrorException e)
    {
        throw new
        HttpRatingServiceNotFound(HttpStatusCode.valueOf(HttpStatus.NOT_FOUND.value()));
    }
}
```

We have created our custom exception, which will be thrown if we catch a Client Exception in the delete Rating Service.

```
HttpRatingServiceNotFound.java
@ResponseStatus(HttpStatus.NOT_FOUND)
public class HotelNotFoundException extends RuntimeException{

    private static final long serialVersionUID = 1L;

    public HotelNotFoundException(String message) {
        super(message);
    }

}
```

In the provided code example, error handling is implemented in the `deleteRating()` method of the `RatingServiceCommunicator` class. It uses try-catch blocks to handle exceptions that may occur during the HTTP request to the Rating service. Specifically, it catches `HttpClientErrorException` to handle client-side errors.

This error handling aims to detect client-side errors during communication with the Rating service, report them, and provide a meaningful response. By catching and throwing specific exceptions, the code ensures that appropriate error messages and status codes are returned to the caller.

Overall, this error-handling approach enhances the robustness of the code by handling potential errors during communication with external services, allowing for better user experience and error resolution strategies.

Message Convertors

In Spring, message converters are crucial in converting Java objects to JSON, XML, and other formats during HTTP requests and responses. The `HttpMessageConverter` interface is used by Spring to perform these conversions. When an HTTP request is made to a Spring application, the message converters come into play to determine the format of the request payload. They convert the request body from its raw form (JSON or XML) into the corresponding Java object.

Similarly, when a Spring application generates an HTTP response, the message converters handle the conversion of the Java object into the desired format specified by the client's request. This allows the response body to be sent in the appropriate format (e.g., JSON or XML) based on the requested content type. Spring provides a variety of built-in message converters that cover common scenarios. These converters can handle data types, such as byte arrays, strings, resources (files), and XML sources. They automatically handle the serialisation and deserialisation of objects based on the content type negotiation and request/response headers.

Types of Message Convertors

By default, the following `HttpMessageConverters` are loaded by Spring. There are several types of message converters available in Spring by default. Here are some commonly used ones:

- **ByteArrayHttpMessageConverter:** This converter handles byte arrays, allowing the conversion of entire Java objects to JSON or other formats and vice versa.
- **StringHttpMessageConverter:** As the name suggests, this converter deals with string conversion. It is responsible for reading and writing strings. When returning a string or integer from a request, this converter deserialises it into the correct form.
- **ResourceHttpMessageConverter:** This converter handles resources and supports byte-range requests. It can read and write resources, making it useful for scenarios involving files or binary data.
- **SourceHttpMessageConverter:** The `SourceHttpMessageConverter` is responsible for handling `Source` objects. It can read and write `Source` objects, typically used in XML-based communication.
- **FormHttpMessageConverter:** This converter specialises in handling HTML forms. It can read and write data from "normal" HTML forms, allowing for seamless interaction with form-based requests.

These message converters are automatically loaded by Spring, and they provide convenient ways to convert Java objects to the desired format during HTTP communication. Spring's message converters greatly simplify handling different data formats and ensure seamless integration with the specified content types.

For additional information on the Message Convertors of `RestTemplate`, please refer to the [MessageConverterDoc](#).

Instructor Codes

- [Hotel Application](#)

References

1. [RestTemplate](#)
2. [getForEntity\(\)](#)
3. [getForObject\(\)](#)
4. [postForObject\(\)](#)
5. [exchange\(\)](#)
6. [Message Converters](#)

Create Query Method

What is a createQuery()?

- We have studied the get and post services and how to use the fundamental operations to create and read data from databases up to this point. However, the method leaves unclear how to obtain the data for all the items and use them for a variety of other tasks.
- To understand this we can use the createQuery() method, an inbuilt method of EntityManager which will be helpful in writing more complex queries.
- JPA EntityManager is used to access a database in a particular application. It is used to manage entity instances, to find entities by their primary key, and to query over all entities.
- createQuery() is a method supported by EntityManager, it creates an instance of Query for executing a (JPQL) Java Persistence query language statement. We will study JPQL queries in further lessons.

```
Session currentSession = entityManager.unwrap(Session.class);

//Pass your query, and name of class on which you want to perform
the query in createQuery method.
Query<Entity> query = currentSession.createQuery("Your
Query",Entity.class);

// Save all item in the list
List<Entity> entitiesList = query.getResultList();
```

Example:

Task: We will create an API in our application to fetch all the Item entities from the database.

ItemController.java

```
package com.cn.ecommerce.controller;

import com.cn.ecommerce.entity.Item;
import com.cn.ecommerce.service.ItemService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api")
public class ItemController {

    @Autowired
    ItemService itemService;

    @GetMapping(path = "/item/{id}")
    public Item getItemById(@PathVariable int id) {
        return itemService.getItemById(id);
    }
}
```

We will create a `getAllItems()` method in `ItemController` class to fetch all item entities.

```
@GetMapping(path = "/item/getAllItems")
Public List<Item> getAllItems() {
    return itemService.getAllItems();
}
```

ItemService.java

```
package com.cn.ecommerce.service;

import com.cn.ecommerce.dao.ItemDAO;
import com.cn.ecommerce.entity.Item;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;

@Service
public class ItemService {

    @Autowired
    private ItemDAO itemDAO;

    @Transactional
    public Item getItemById(int id) {
        return itemDAO.get(id);
    }
}
```

We will now implement the service called by the controller in ItemService class.

```
@Transactional
public List<Item> getItem(){
    return itemDAO.get();
}
```

ItemDAOImpl.java

```
package com.cn.ecommerce.dao;

import com.cn.ecommerce.entity.Item;
import com.cn.ecommerce.entity.ItemDetails;
import org.hibernate.Session;
import org.hibernate.query.Query;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;

import javax.persistence.EntityManager;
import java.util.List;

@Repository
public class ItemDAOImpl implements ItemDAO{

    @Autowired
    private EntityManager entityManager;

    @Override
    public Item get(int id) {
        Session currentSession = entityManager.unwrap(Session.class);
        Item item = currentSession.get(Item.class, id);
        return item;
    }
}
```

In the ItemDAOImpl class we will use createQuery() method to fetch all item entities from the database.

```
@Override
public List<Item> get() {
    Session currentSession = entityManager.unwrap(Session.class);
    Query<Item> query = currentSession.createQuery("from Item",
Item.class);
    List<Item> list = query.getResultList();
    return list;
}
```

What's more to learn?

So, till now we have learned to use the `createQuery()` method to fetch all entities from the database, it can also be used to create dynamic queries. We can also pass variables in our queries and perform more operations:

- Update and delete operations on entities.
- Using aggregate functions like `sum()`, `max()` and `min()`.
- Fetching records with pagination, like fetching records from 10th to 20th number.

To know more about their implementation, you can refer to this [link](#).

Introduction to Hibernate and CRUD [3199]

What is JDBC?

JDBC stands for Java Database Connectivity, which provides low-level, database-dependent connectivity between the Java programming language and a wide range of databases.

Advantages of JDBC:

- Send queries and update statements to the database.
- Enables a Java application to communicate with a database.
- Retrieve and process the results received from the database in answer to your query.

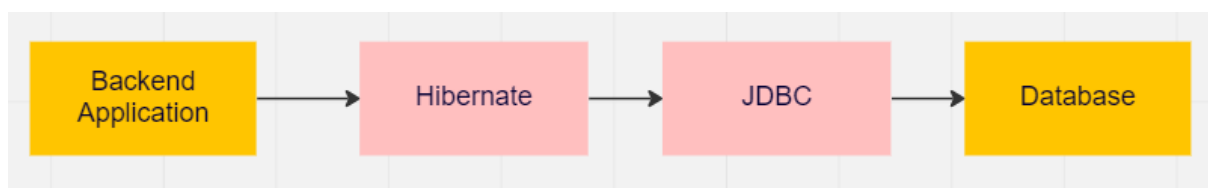
What is Hibernate in JAVA?

Hibernate provides a standardized and efficient way of storing and retrieving data in a relational database using Java. It facilitates mapping Java objects to database tables through **Object-Relational Mapping (ORM)**. Hibernate implements the **Java Persistence API (JPA)** specification, which sets standards for persisting data in Java applications.

What is Persistence?

Persistence refers to the ability of data to be stored and retrieved from a database even after the Java application that created it has been closed. Persistence is achieved through various technologies such as ORM, JPA, and frameworks like Hibernate.

Working of Hibernate



When an application requests data from the database, Hibernate retrieves the data using JDBC and maps it to **Java objects**, which are then returned to the application. When an application saves data to the database, Hibernate takes the Java objects, maps them to database tables, and persists the data using JDBC.

Overall, Hibernate simplifies database interactions by providing a higher-level API and abstracting the low-level details of JDBC and SQL.

What is the need for Hibernate?

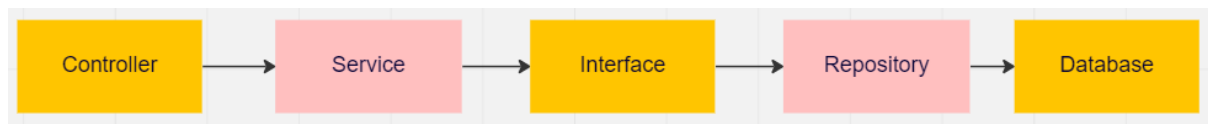
Some of the advantages of Hibernate are

1. Hibernate offers automated and **efficient mapping** of Java objects to database tables through ORM.
2. It also generates **database-independent queries**. So you don't need to write database-specific queries like you will have the same query for MySQL database as for PostgreSQL Database.
3. It simplifies database operations and reduces the boilerplate code needed for database interactions, making it easier to develop and maintain database-driven applications.
- 4.

Hibernate Annotations

These annotations define the mapping between Java objects and database tables and are placed on the Java classes, fields, and methods.

Components of a SpringBoot Application



The flow of components in a SpringBoot application is as follows:

Controller: Receives a request from the user, passes it to the service layer for processing, and returns the response to the user.

Service: Processes the request from the controller, interacts with the repository to retrieve or persist data, and returns the result to the controller.

Repository: Acts as an interface to the database, performing operations such as retrieving, updating, and deleting data and returning the result to the service layer.

Interface: Defines the methods and properties the repository and service layer must implement, serving as a blueprint for these components.

Database: Persists the data for the application and provides a way for the repository to store and retrieve data.

We will now discuss the different annotations used across the four layers of a SpringBoot Application, i.e., Model, Service, Controller, and DAL Layer.

Database: Model Layer

Model layer is the part of the application that interacts with the database using Hibernate. It comprises Java objects representing the application's data entities, such as items, orders, or products. These objects are known as **entities**.

We will now go through the various annotations used in the Model layer to map a class in java to a table in the database.

@Entity

The @Entity annotation in Hibernate indicates that a class represents a database table. When Hibernate encounters an @Entity annotated class, it maps the class and its properties to a database table with the same name as the class.

```
package com.cn.cnkart.entity;
import javax.persistence.*;
@Entity
public class Item {
}
```

@Id

The @Id annotation in Hibernate is used to indicate the primary key column of a database table. The primary key is a unique identifier for each record in a database table and is used to locate a specific record.

For example

```
import javax.persistence.Id;
@Entity
public class Item {
    @Id
    private int id;
}
```

In this example, the id field is annotated with @Id, indicating that it represents the primary key column of the "Item" table in the database.

@GeneratedValue & @GenerationType

The @GeneratedValue annotation in Hibernate automatically generates **unique values** for a primary key column in a database table. It is used in conjunction with the @Id annotation.

Generation strategies can use four @GenerationType:

- **AUTO**: Hibernate selects the strategy based on the used dialect.

- **IDENTITY**: Hibernate relies on an auto-incremented database column to generate the primary key,
- **SEQUENCE**: Hibernate requests the primary key value from a database sequence.
- **TABLE**: Hibernate uses a database table to simulate a sequence.

For example,

```
import javax.persistence.*;
import javax.persistence.*;
@Entity
public class Item {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
}
```

In this example, `@GeneratedValue` indicates that Hibernate should automatically generate values of the id column.

@Column

The `@Column` annotation in Hibernate maps a **column** in a Java class to a field in a database table. The annotation is used to specify the column's name in the database table.

```
package com.cn.cnkart.entity;
import javax.persistence.*;
@Entity
public class Item {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column
    private int id;
    @Column(name="item_name")
    private String name;
    @Column(name="item_description")
    private String description;
}
```

The name field is annotated with `@Column`, indicating that it should be mapped to a column in the "Item" table in the database. The annotation specifies the name of the column as "item_name."

Note: The `@Column` annotation is optional; if not specified, Hibernate will use the field's name as the column's name.

Controller Layer

Controller layer is responsible for receiving user requests and coordinating with the Hibernate layer to perform database operations, for example, creating, reading, updating, or

deleting records in the database. The Controller layer acts as a mediator between the user interface and the database.

@RestController

This annotation indicates that this class is a Spring controller and is responsible for handling RESTful web requests. It is a combination of two annotations:

1. **@Controller** - This @Controller annotation marks the class as a controller class.
2. **@ResponseBody** - This annotation indicates that the response from the controller method should be sent to the client as the body of the HTTP response.

For example,

```
package com.cn.cnkart.controller;

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

@RestController
public class ItemController {
}
```

@RequestMapping

This annotation is used to map HTTP requests to handler methods of MVC and REST controllers. To configure the mapping of web requests, we use the @RequestMapping annotation. The @RequestMapping annotation is applied to the controller class.

For example,

```
package com.cn.cnkart.controller;

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

@RestController
@RequestMapping("/item")
public class ItemController {
}
```

@RequestMapping maps HTTP requests to the "/item" URL to the methods in this class. This class's methods will handle requests that start with "/item."

@GetMapping

@GetMapping annotation maps HTTP GET requests onto specific handler methods. It is a composed annotation that acts as a shortcut for **@RequestMapping(method = RequestMethod.GET)**. The @GetMapping annotated methods handle the HTTP GET requests matched with the given URI expression.

For example,

```
package com.cn.cnkart.controller;

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

@RestController
@RequestMapping("/item")
public class ItemController {
    @GetMapping("/id/{id}")
    public Item getItemById(@PathVariable int id) {
    }
}
```

Here, `@GetMapping` maps the HTTP GET request with the specified URL pattern to this method. The pattern `"id/{id}"` includes a path variable `{id}`, which represents a placeholder for a dynamic value that can be passed in the request URL.

The `@PathVariable` annotation tells Spring to extract the value of the `id` path variable from the URL and pass it as a parameter to the method.

@PostMapping

The `@PostMapping` is a specialized version of `@RequestMapping` annotation that acts as a shortcut for `@RequestMapping(method = RequestMethod.POST)`. The `@PostMapping` annotated methods handle the HTTP POST requests matched with the given URI expression.

For example,

```
package com.cn.cnkart.controller;

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

@RestController
@RequestMapping("/item")
public class ItemController {
    @PostMapping("/save")
    public void saveItem(@RequestBody Item item){
    }
}
```

`@PostMapping` annotation maps the HTTP POST request with the specified URL pattern `"save"` to this method. The `@RequestBody` annotation tells Spring to extract the request body of the incoming HTTP POST request and map it to the `item` parameter.

@Delete Mapping

The `@DeleteMapping` is a specialized version of `@DeleteMapping` annotation that acts as a shortcut for `@RequestMapping(method = RequestMethod.DELETE)`. The `@DeleteMapping` annotated methods handle the HTTP DELETE requests matched with the URI expression.

For example,

```
package com.cn.cnkart.controller;

import org.springframework.web.bind.annotation.*;
@RestController
@RequestMapping("/item")
public class ItemController {
    @DeleteMapping("/delete/id/{id}")
    public void deleteItem(@PathVariable int id){
    }
}
```

`@DeleteMapping` annotation maps the HTTP DELETE request with the specified URL pattern `/delete/id/{id}` to this method. The pattern `/delete/id/{id}` includes a path variable `{id}`, which represents a placeholder for a dynamic value that can be passed in the request URL. `@PathVariable` annotation tells Spring to extract the value of the `id` path variable from the URL and pass it as a parameter to the method.

@Put Mapping

The `@PutMapping` is a specialized version of `@PutMapping` annotation that acts as a shortcut for `@RequestMapping(method = RequestMethod.PUT)`. The `@PutMapping` annotated methods handle the HTTP PUT requests matched with the given URI expression.

For example,

```
package com.cn.cnkart.controller;

import org.springframework.web.bind.annotation.*;
@RestController
@RequestMapping("/item")
public class ItemController {
    @PutMapping("/update")
    public void updateItem(@RequestBody Item updateItem)
    {
        itemService.update(updateItem);
    }
}
```

`@PutMapping` annotation maps the HTTP PUT request with the specified URL pattern `"/update"` to this method. The `@RequestBody` annotation tells Spring to extract the request body of the incoming HTTP POST request and map it to the `item` parameter.

Service Layer

In Hibernate, the service layer manages **the persistence** of data to and from the database using Hibernate's ORM capabilities. The service layer acts as a bridge between the database and the application's business logic.

The service layer provides logic to operate on the data sent to and from the DAO and the client. Another user of the service layer is **security**, and if you provide a service layer that has no relation to the DB, then it is more difficult to gain access to the DB from the client except through the service.

For example,

```
package com.cn.cnkart.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import com.cn.cnkart.service.ItemService;

@RestController
@RequestMapping("/item")
public class ItemController {

    @Autowired
    ItemService itemService;

    @GetMapping("/id/{id}")
    public Item getItemById(@PathVariable int id) {
        return itemService.getItemById(id);
    }
}
```

`@Autowired` annotation to injects an instance of `ItemService` into the controller. `ItemService` object calls its `getItemById` method, which is now responsible for retrieving an item from some persistent storage, such as an `Item` entity by its `id`.

@Service

The `@Service` annotation is a marker annotation in the SpringBoot framework that indicates that a class is a service component. By annotating a class with `@Service`, you indicate to Spring that it should consider the class as a candidate for component scanning.

For example,

```
package com.cn.cnkart.service;

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

@Service
public class ItemService {
}
```

@Transactional

This annotation is used to declare a method or class as transactional in the SpringBoot framework. By annotating a method or class with `@Transactional`, you indicate to Spring that the operations performed within the method or class should be executed as a single atomic transaction.

This means that either all operations will be executed and committed to the database, or none will be executed, and the database will be rolled back to its previous state.

For example,

```
package com.cn.cnkart.service;
import javax.transaction.Transactional;
import org.springframework.beans.factory.annotation.*;

@Service
public class ItemService {
    @Transactional
    public Item getItemById(int id) {
        return itemDAL.getById(id);
    }
}
```

In this example, when the “getItemById” method is called, it will be executed within the scope of a database transaction, ensuring that If any of the database operations fail, the entire transaction will be **rolled back**. The database will be returned to its previous state.

Interface: DAL Layer

The Data Access Layer (DAL) in Hibernate refers to the components responsible for communicating with the database to perform CRUD (Create, Read, Update, and Delete) operations.

We will create the interface for DAL Layer,

```
package com.cn.cnkart.dal;
```



```
import com.cn.cnkart.entity.Item;

public interface ItemDAL {

    Item getById(int id);

    void save(Item item);

    void delete(int id);

    void update(Item updateItem);

}
```

Repository: DAL Implementation Layer

The implementation class of DAL Interface typically contains the classes and methods that handle the mapping of objects to database tables, the execution of database queries, and the management of transactions.

@Repository

The annotation indicates that a class is a repository, a component that provides access to the data stored in a database.

For example,

```
package com.cn.cnkart.dal;

import org.springframework.stereotype.Repository;

@Repository
public class ItemDALImpl implements ItemDAL{

}
```

Entity Manager

Is used to create **sessions** that interact with a database. The Entity Manager helps you manage interactions with the database by providing methods for performing database operations, creating queries, and managing transactions. The sessions created by the Entity Manager allow you to perform database operations consistently and reliably.

Session Methods: GET

```
package com.cn.cnkart.dal;
```

```

import javax.persistence.EntityManager;
import org.hibernate.Session;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import com.cn.cnkart.entity.Item;

@Repository
public class ItemDALImpl implements ItemDAL{
    @Autowired
    EntityManager entityManager;

    @Override
    public Item getById(int id) {
        Session session = entityManager.unwrap(Session.class);
        Item item = session.get(Item.class, id);
        return item;
    }
}

```

The `EntityManager` is injected into the class using the `@Autowired` annotation. This annotation tells Spring to automatically create an instance of the `EntityManager` and inject it into the `entityManager` field.

The “unwrap” method of `EntityManager` returns the underlying **Hibernate Session** from the `EntityManager`, which can be used to perform additional operations not available through `EntityManager`. Once the `Hibernate Session` is obtained, the `get` method is called on the `Session`, passing in the “**Item.class**” and the object's id to retrieve. This method retrieves the `Item` object with the specified id from the database and returns it.

Session Methods: GET-ALL

```

@Override
public List<Item> getAllItems() {
    Session session = entityManager.unwrap(Session.class);
    //Pass your query, and name of class on which you want to perform the query
    in createQuery method.
    Query<Entity> query = session.createQuery("from Item",Entity.class);
    // Save all item in the list
    List<Item> itemList = query.getResultList();
    return itemList;
}

```

To fetch all `Items` from the database we can use `createQuery()` method. To learn more about this method, you can refer to this [link](#).

Session Methods: POST

```
@Override
public void save(Item item) {
    Session session = entityManager.unwrap(Session.class);
    session.save(item);
}
```

Similarly, you can use the “**session.save(item)**” method to save an Item entity into the database.

Session Methods: DELETE

```
@Override
public void delete(int id) {
    Session session = entityManager.unwrap(Session.class);
    Item item = session.get(Item.class, id);
    session.delete(item);
}
```

The Entity Manager is used to obtain a Hibernate Session, which is then used to retrieve the Item object from the database. Then, you can use the “**session.delete(item)**” method to delete the Item entity from the database.

Session Methods: UPDATE

```
@Override
public void update(Item updateItem) {
    Session session = entityManager.unwrap(Session.class);
    //fetch the current item details from the DB
    Item currentItem = session.get(Item.class, updateItem.getId());

    //update the details in the current object
    currentItem.setDescription(updateItem.getDescription());
    currentItem.setName(updateItem.getName());
    //update the database
    session.update(currentItem);
}
```

The Entity Manager is used to obtain a Hibernate Session, which is then used to update the Item object from the database. First, the “get” method retrieves the current Item object with the specified id from the database. Finally, the update method is called on the Session, passing in the current Item object. The “**session.update(currentItem)**” method updates the Item object in the database with the new values.

Note: When you call the update method, Hibernate will synchronize the state of the persistent object with the database, updating the database record to reflect any changes made to the object. For this to work, the object must already be associated with a Hibernate session and have a corresponding database record.

Conclusion

In this lesson, we covered the basics of Hibernate and JDBC. We learned about the different layers of a Spring Boot application and the various annotations used in each layer. We also discussed the concept of sessions and how they can be used to perform CRUD (Create, Read, Update, and Delete) operations on a database.

Instructor Codes

- [CNkart Application](#)

References

1. [JDBC](#)
2. [Hibernate](#)
3. [Hibernate Session Methods](#)

Introduction to Hibernate and Relationships

Hibernate Relationships

Hibernate allows developers to map Java objects to database tables and manipulate them using object-oriented programming concepts. One crucial aspect of developing Hibernate applications is dealing with relationships between objects and tables.

There are three types of relationships in Hibernate:

1. One To One Relationship
2. One To Many Relationships
3. Many To Many Relationships

One To One Relationship

In a one-to-one relationship, one entity is associated with one instance of another entity. An example of a source entity can be, at most, mapped to one instance of the target entity.

There can be two types of One To One Relationships:

1. Unidirectional Relationship

Unidirectional one-to-one Relationship is a type of Relationship where one entity is associated with exactly one instance of another. Still, while one entity is linked to another, the related entity does not necessarily have a corresponding link to the first entity.

For example, "Item" refers to a product or item being sold or managed within a system. In contrast, "ItemDetails" refer to additional information about that item, such as its name, description, price, and other details. The relationship between Item and Item Details is one-way, and Item Details only exist to provide more information about a specific Item.

Let's go through the code:

- The "@OneToOne" annotation specifies that there is a one-to-one relationship between the "Item" entity and the "ItemDetails" entity.

- The cascading attribute specifies that any changes made to the "Item" entity (e.g., deleting an "Item") will be cascaded to the associated "ItemDetails" entity so that it will be deleted as well.
- The getters and setters of "ItemDetails" are essential when we want to fetch and update "ItemDetails" associated with an "Item".

```
//ItemEntity
@Entity
@Table(name="item")
public class Item {

    @OneToOne(cascade = CascadeType.ALL)
    private ItemDetails itemDetails;

    //Getters and setters of ItemDetails
}
```

2. Bidirectional Relationship

A Bidirectional one-to-one Relationship refers to a relationship between two entities where each entity has a reference to the other. Meaning there is an "OneToOne" relationship between two entities in which each entity can access the other. Considering the above example of an Item entity and an ItemDetails entity, we will need to implement a bidirectional one-to-one Relationship if we want to fetch details of the Item entity of a specific ItemDetail entity.

Let's review the code:

1. We must write additional code in ItemDetails Entity to link it back to Item Entity.
2. Mapped by specifies the bidirectional relationship between ItemDetails and Item Entities. Its value indicates the variable we want to link with from the Item entity.
3. We are using "CascadeType.ALL", which means that any changes made to the "ItemDetails" entity will be cascaded to the associated "Item" entity.

```
//Item Details Entity
@Entity
@Table(name="item_details")
public class ItemDetails {

    @OneToOne(mappedBy = "itemDetails", cascade = CascadeType.ALL)
    private Item item;

}
```

One To Many Relationship

In a one-to-many relationship, one entity is associated with multiple instances of another entity. For example, consider an Item entity and an ItemReview entity. An Item may have multiple ItemReviews associated with it, and an ItemReview can only be associated with one Item. To map this relationship in Hibernate, we can add a **@OneToMany** annotation to the Item entity, specifying the ItemReview entity as the target of the relationship.

There can be two types of One To Many Relationships:

1. Unidirectional Relationship

In a unidirectional one-to-many relationship, a single entity on one side of the relationship is associated with multiple instances of another entity on the other side of the relationship, but the other entity does not have a direct reference back to the first entity.

For example, the Item entity can be associated with multiple instances of the ItemReview entity. However, the ItemReview entity does not have a direct reference back to the Item entity.

Let's go through the code:

- To implement this Relationship, we would typically add a List or Set the property to the Item entity to hold all the associated ItemReview entities.
- We will annotate this property with the **@OneToMany** annotation, which tells that this property represents a one-to-many relationship.
- The cascade attribute specifies that any changes made to the Item entity should also be applied to its associated ItemReview entities (e.g., if an Item is deleted, all its associated ItemReview entities should also be deleted).
- The JoinColumn annotation specifies the column in the ItemReview table.

```
//ItemEntity
@Entity
@Table(name="item")
public class Item {

    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name="item_id")
    private List<ItemReview> itemReview;

    //Getters and setters of ItemReview

}
```

2. Bidirectional Relationship

In a bidirectional one-to-many Relationship, one entity has a one-to-many relationship with another entity, and the other entity has a many-to-one relationship back to the first entity. Meaning that the "one" entity can have multiple related "many" entities, and each "many" entity can have a direct reference back to the associated "one" entity.

Considering the above example of an Item entity and an ItemReview entity, we must implement a bidirectional one-to-one Relationship to fetch the Item of the ItemReview entity of a specific ItemReview entity.

Let's go through the code:

- The mappedBy attribute in the @OneToMany annotation indicates that the ItemReview entity manages the relationship and will use the item property in the ItemReview entity as the foreign key to the Item entity.
- The cascade attribute specifies that any changes made to the Item entity should also be applied to its associated ItemReview entities (e.g., if an Item is deleted, all its associated ItemReview entities should also be deleted).

```
//ItemEntity
@Entity
@Table(name="item")
public class Item {
    @OneToMany(mappedBy="item",cascade = CascadeType.ALL)
    @JsonManagedReference
    private List<ItemReview> itemReview;

    //Getters and setters of ItemReview
}
```

- The @ManyToOne annotation indicates that each ItemReview entity is associated with a single Item entity
- The @JoinColumn annotation specifies the column in the ItemReview table that contains the foreign key to the Item table.

```
//Item Details Entity
@Entity
@Table(name="item_review")
public class ItemReview {

    @ManyToOne
    @JoinColumn(name="item_id")
    @JsonBackReference
    private Item item;
}
```


Many To Many Relationship

In a many-to-many relationship, multiple instances of one entity are associated with multiple instances of another entity. For example, consider an Item entity and an Order entity. An Item may be associated with multiple Orders, and an Order may be associated with multiple Items. To map this relationship in Hibernate, we can add a `@ManyToMany` annotation to both the Item and Order entities, specifying the other entity as the target of the relationship. This will create a third table, a join table, that stores the associations between the two entities.

Let's review the code:

- The `@ManyToMany` annotation defines the relationship between Order and Item.
- The cascade attribute specifies the cascade behavior for any operations performed on the association. We don't want to delete any item when we delete an order so we won't use `CascadeType.REMOVE`, but deleting this order will delete the relationship between this order and all associated items from the `order_item` table.
- The relationship between these entities is established using a join table named `order_item`, which has foreign key columns referencing the primary keys of the Order and Item tables.
- The `@JoinTable` annotation specifies the name of the join table and the foreign key columns used to establish the relationship between the Order and Item entities. The `joinColumns` attribute specifies the column used to reference the Order table, while the `inverseJoinColumns` attribute specifies the column used to reference the Item table.

```
//OrderService
@Entity
@Table(name="orders")
public class Order {

    @ManyToMany(
        cascade={CascadeType.MERGE,CascadeType.PERSIST,CascadeType.REFRESH}
    )
    @JoinTable(name="order_item",
        joinColumns =@JoinColumn(name="order_id"),
        inverseJoinColumns = @JoinColumn(name="item_id"))
    private List<Item> items;
}
```

- The `@ManyToMany` annotation is used to define the relationship between Item and Order.
- The `mappedBy` attribute specifies that the relationship is mapped by the items property on the Order entity. This means that the join table and the foreign key columns are defined on the Order entity rather than the Item entity.

```
//Item Entity
@Entity
@Table(name="item")
public class Item {

    @ManyToMany(mappedBy = "items")
    @JsonIgnore
    private List<Order> orders;
}
```

- **Note 1:** The **@JsonIgnore** annotation is used to ignore the order's property when the Item entity is serialized to JSON, preventing a possible infinite circular reference between the Order and Item entities.
- **Note 2:** Before saving the order into the database, fetch all the item values and set those values to your new order.

```
//Order Service
@Transactional
public void saveOrder(Order order) {

    Order saveOrder =new Order();
    saveOrder.setOrderType(order.getOrderType());
    List<Item> itemList = new ArrayList<>();
    for(Item item:order.getItems()) {
        Item currentItem=itemDAL.getById(item.getId());
        itemList.add(currentItem);
    }
    saveOrder.setItems(itemList);
    orderDAL.save(saveOrder);
}
```

What is H2 Database?

H2 Database is an open-source, in-memory relational database management system written in Java. It is designed to be fast, lightweight, and easy to use, making it an ideal choice for testing and development.

In-memory databases store data in the computer's memory rather than on a disk. This means the database is created and stored in the computer's memory and lost when the program or application is closed. This makes it ideal for testing and development because it is easy to set up and use.

- The H2 database can be easily integrated with Java applications by adding the following dependency to the project's build file:
After adding the dependency, you can configure the database connection properties in the application's configuration file, like the following example for a Spring Boot application:

```
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <scope>runtime</scope>
</dependency>
```

- The configuration file sets up an H2 database with an in-memory storage model.
- The JDBC URL specifies that the H2 database is in memory and has the name **"testdb"**.
- The driver class is a Java class that implements the JDBC API and provides connectivity to the database. In this case, we have specified the JDBC driver for the H2 database.
- Setting Hibernate dialect here for the H2 database is similar to setting the dialect for the MySQL database.
- The Hibernate ddl-auto property is set to **"create"**, meaning the database schema will be created automatically when the application starts.
- The H2 console can be enabled by setting the **Console.enabled** property to true in the application configuration, this allows access to the H2 database console and interaction with the database during runtime.

```
spring:
  datasource:
    url: jdbc:h2:mem:testdb
    username: sa
    password: password
    driverClassName: org.h2.Driver
  jpa:
    spring.jpa.database-platform: org.hibernate.dialect.H2Dialect
    Hibernate.ddl-auto: create
```

Conclusion

In this lesson, we covered the basics of Hibernate and Hibernate. We discussed different types of relationships in Hibernate, including one-to-one, one-to-many, and many-to-many relationships. We also saw examples of unidirectional and bidirectional Relationships for these relationships. Furthermore, we discussed using the H2 database for testing and in-memory database operations and how to configure Hibernate with the H2 database.

Instructor Codes

- [CNKart Application](#)

References

1. [Hibernate Relationships](#)
2. [H2 Database](#)

More On Hibernate & Relationships

Hibernate

Hibernate is an object-relational mapping (ORM) framework that simplifies database interactions in Java applications. It allows developers to map Java objects to database tables and perform CRUD (Create, Read, Update, Delete) operations without writing complex SQL queries.

Relationships in Hibernate

1. One-to-One Relationship: A one-to-One relationship in database design represents a connection between two tables where one record in the first table is related to exactly one record in the second table, and vice versa. This relationship means that for each record in one table, there is at most one related record in the other table.

Characteristics of a One-to-One Relationship:

- **Single Connection:** Each record in one table is associated with, at most, one record in the other table.
- **Unique Correspondence:** There's a unique correspondence between the records of both tables.
- **Foreign Key:** Typically, one table holds a foreign key that references the primary key of the other table.

Example: Consider two entities: Person and Passport. In a one-to-one relationship, one person has exactly one passport, and each passport belongs to precisely one person. In database terms, it means that a Person entity can be associated with at most one Passport entity, and vice versa.

Example Code (Using Java and JPA/Hibernate Annotations)

Here's an example illustrating a One-to-One relationship between *Person* and *Passport* entities using JPA/Hibernate annotations:

Person entity

```
@Entity
public class Person {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @OneToOne(mappedBy = "person")
    private Passport passport;

    // other fields, getters, and setters
}
```

Passport entity

```
@Entity
public class Passport {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String passportNumber;

    @OneToOne
    @JoinColumn(name = "person_id")
    private Person person;

    // other fields, getters, and setters
}
```

- In this example, Person and Passport are two entities. The Person entity holds a reference to the Passport entity using the **@OneToOne** annotation, and the Passport entity has a reference to the Person entity using the same annotation.
- In Java Persistence API (JPA), **@Entity** is an annotation that marks a Java class as a persistent entity.
- When you mark a Java class with **@Entity**, you're telling the JPA provider (like Hibernate) that instances of this class should be treated as entities that can be persisted in a database. Each instance of an entity class typically corresponds to a row in a database table.

- The **@Id** annotation marks a field in an entity class as the primary key of the corresponding database table. It signifies that this field uniquely identifies each record in the table.
- **GenerationType.IDENTITY**: Indicates that the database will automatically generate unique primary key values. This strategy is typically used with auto-incremented columns in databases like MySQL, PostgreSQL, etc.
- The **@JoinColumn** annotation is used to specify the foreign key column in the Passport table (person_id) that references the primary key of the Person table.

This relationship setup allows a Person to have exactly one Passport, and a Passport to be associated with exactly one Person, demonstrating a One-to-One relationship between these entities.

2. **One-to-Many Relationship**: A One-to-Many relationship in database design refers to a scenario where one entity instance is associated with multiple instances of another entity. This relationship is a fundamental concept in relational databases and is commonly used in various applications.

Characteristics of One-to-Many Relationship:

One Side

- Refers to the entity that stands alone in the relationship.
- In a One-to-Many relationship, this entity can exist independently, with multiple instances of another entity associated with it.
- For example, the Author entity stands on one side in a scenario involving the Author and Book. An author can exist without any books or have multiple books associated with them.

Many Side

- Refers to the entity with multiple instances associated with a single instance of the other entity.
- This side of the relationship represents the entity dependent on or associated with the entity on the one side.
- In the Author and Book example, the Book entity represents the many side. Multiple books can be associated with a single author.

Example:

Consider entities **Author** and **Book**:

- **Author:** An author can write multiple books.
- **Book:** Each book is written by exactly one author.

Implementation in Hibernate/Spring Boot:

Author class

```
@Entity
public class Author {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @OneToMany(mappedBy = "author", cascade = CascadeType.ALL)
    private List<Book> books;

    // Constructors, getters, setters
}
```

Book class

```
@Entity
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String title;

    @ManyToOne
    @JoinColumn(name = "author_id")
    private Author author;

    // Constructors, getters, setters
}
```

Author Entity: It has a one-to-many relationship with the Book entity. An Author can have multiple Book instances associated with it.

Book Entity: It has a many-to-one relationship with the Author entity. Each Book belongs to exactly one Author.

Explanation:

- **@OneToMany** annotation in the Author class represents the one-to-many relationship. The `mappedBy` attribute specifies the field in the Book entity that owns the relationship. In this case, it's the `author` field in the Book class.
- **@ManyToOne** annotation in the Book class defines the many-to-one side of the relationship. The `JoinColumn` annotation is used to specify the foreign key column (**`author_id`**) in the Book table that references the Author table.

Important Points:

- The **`cascade = CascadeType.ALL`** attributes ensure that operations performed on the Author entity (e.g., saving, updating, deleting) will cascade to associated Book entities.
- In the database schema, the Book table will contain a foreign key column (**`author_id`**) referencing the `id` column in the Author table, establishing the relationship.
- This setup allows you to navigate from an Author instance to its associated Book instances and vice versa. It enables effective management of one-to-many relationships in your Spring Boot application using Hibernate.

3. **Many-to-Many**: A Many-to-Many relationship in database design represents a scenario where multiple instances of one entity are associated with multiple instances of another entity. Implementing such relationships requires an intermediary table (a junction or link table) to establish connections between entities in relational databases.

Characteristics of Many-to-Many Relationship:

- Exists between two entities, A and B.
- Multiple instances of entity A can be linked to multiple instances of entity B, and vice versa.
- An intermediary table establishes and manages connections between the two entities.

Example using Students and Courses:

Student Entity:

```
@Entity
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
```

```
@ManyToMany
@JoinTable(
    name = "student_course", // Intermediary table name
    joinColumns = @JoinColumn(name = "student_id"),
    inverseJoinColumns = @JoinColumn(name = "course_id")
)
private List<Course> courses;

// Constructors, getters, setters
}
```

Course Entity:

```
@Entity
public class Course {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @ManyToMany(mappedBy = "courses")
    private List<Student> students;

    // Constructors, getters, setters
}
```

Explanation:

- **Student Entity:**

@ManyToMany annotation establishes the Many-to-Many relationship with the Course entity.

@JoinTable specifies the name of the intermediary table (student_course) and the columns that link the Student and Course entities.

Course Entity:

@ManyToMany is specified in the Student entity, so in the Course entity, mappedBy = "courses" indicates the field (courses) in the Student entity that owns the relationship.

Intermediary Table:

- The `student_course` table (intermediary table) contains foreign key columns referencing the id columns of both the Student and Course tables.
- This table establishes connections between students and courses, allowing many students to enrol in many courses without any entity owning the relationship directly.

Illustration:

- A Student can be enrolled in multiple Course instances (e.g., John is enrolled in Math, Science, and English courses).
- A Course can have multiple Student instances enrolled (e.g., the Math course has John, Alice, and Sarah enrolled).

This Many-to-Many relationship allows for the flexibility of multiple students being associated with multiple courses and vice versa, managed through an intermediary table that facilitates these connections in a Spring Boot application using Hibernate.

Overview of JPA/Hibernate Cascade Types

Cascade types in Java Persistence API (**JPA**) with Hibernate define how entity state changes should propagate from one entity to other associated entities. These cascade types specify whether operations performed on an entity should be cascaded (applied) to its related entities.

Hibernate provides various cascade types, which can be specified using annotations to manage entity state transitions automatically.

Here's an overview of common cascade types:

CascadeType Overview:

1. **ALL:** All operations are cascaded, including persist, merge, remove, refresh, and detach. This effectively means any operation on the parent entity will propagate to its associated entities.
2. **PERSIST:** When a new entity is persisted (saved), the operation is cascaded to associated entities, making them persist.
3. **MERGE:** When an entity is merged (updated), the operation cascades to associated entities, merging their state with the current persistence context.
4. **REMOVE:** When an entity is removed (deleted), the operation cascades to associated entities, removing them as well.
5. **REFRESH:** Refreshing an entity state from the database will also cascade to associated entities, refreshing their state.

6. **DETACH:** Detaching an entity from the persistence context will also cascade to associated entities, detaching them.

Example Usage:

```
@Entity
public class Author {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @OneToMany(mappedBy = "author", cascade = CascadeType.ALL)
    private List<Book> books;

    // Constructors, getters, setters
}
```

In this example:

`**cascade = CascadeType.ALL**` on the **books** field of the **Author** entity indicates that any operation (persist, merge, remove, etc.) performed on an **Author** entity will cascade to its associated **Book** entities.

Notes:

- Cascade types are specified at the relationship level in JPA annotations, allowing for fine-grained control over how entity state transitions are managed.
- Care should be taken when using cascade types to avoid unintended side effects, such as inadvertently removing associated entities.

Understanding and appropriately utilising cascade types in JPA/Hibernate can significantly simplify the management of entity state transitions and the associated entity graph. It ensures consistency and helps in reducing boilerplate code for handling related entities' state changes.

Eager/Lazy Loading in Hibernate

Overview

When working with an ORM, data fetching/loading can be classified into eager and lazy. This quick tutorial will point out differences and show how we can use these in Hibernate.

Eager and Lazy Loading

- **Eager Loading** is a design pattern in which data initialisation occurs on the spot.
- **Lazy loading** is a design pattern that we use to defer the initialisation of an object as long as possible.

Eager Loading: Eager Loading is a strategy where the associated objects are fetched from the database along with the main entity. In other words, when you load an entity, Hibernate also loads its related entities immediately.

For example, consider two entities, Author and Book, where an author can have multiple books. With eager loading, Hibernate will automatically fetch all associated Book entities when you fetch an Author entity.

Here's a simple code snippet using Hibernate annotations to demonstrate eager loading:

Author entity

```
@Entity
public class Author {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @OneToMany(mappedBy = "author", fetch = FetchType.EAGER)
    private List<Book> books;

    // Getters and setters
}
```

Book entity

```
@Entity
public class Book {
    @Id
```

```
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;

private String title;

@ManyToOne
@JoinColumn(name = "author_id")
private Author author;

// Getters and setters
}
```

In this example, the **@OneToMany** relationship in the **Author** entity specifies **FetchType.EAGER** indicates that whenever an **Author** object is fetched, its associated **Book** objects will also be fetched eagerly.

Lazy Loading: Lazy Loading is a strategy where associated objects are not fetched from the database immediately when the main entity is loaded. Instead, they are loaded only when accessed or requested.

Using the same Author and Book entities:

```
@Entity
public class Author {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @OneToMany(mappedBy = "author", fetch = FetchType.LAZY)
    private List<Book> books;

    // Getters and setters
}
```

Here, the **@OneToMany** relationship in the **Author** entity specifies **FetchType.LAZY** indicates that the associated **Book** objects will not be loaded immediately when an **Author** object is fetched. They will only be loaded from the database when the **books** list is accessed.

Lazy Loading can help optimise performance by reducing unnecessary database queries. However, developers need to be cautious about potential **LazyInitializationExceptions** that may occur if the associated objects are accessed outside the session where they were initially loaded. These strategies provide flexibility in fetching associated objects in Hibernate, allowing you to optimise performance based on your application's requirements.

@JoinColumn

In Hibernate, **@JoinColumn** defines the owning side of a relationship between two entities. It specifies the column in the database that maintains the association between these entities.

For example, let's consider an Author entity and a Book entity. An author can have multiple books, creating a one-to-many relationship between them.

Author entity

```
@Entity
public class Author {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @OneToMany
    @JoinColumn(name = "author_id") // Defines the foreign key column in
the Book table
    private List<Book> books;

    // Getters and setters
}
```

Book entity

```
@Entity
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String title;

    @ManyToOne
    private Author author; // No need for @JoinColumn here, as it's
already defined in Author

    // Getters and setters
}
```

In this example, the **@JoinColumn** annotation is used in the **Author** entity to specify that the **author_id** column in the **Book** table will maintain the relationship with the **Author** table. It defines the foreign key column in the **Book** table that links it to the **Author** table.

mappedBy

The `mappedBy` attribute is used on the inverse side of a bidirectional relationship to specify the field that owns the relationship. It indicates that the field maps the relationship in the owning entity.

In the same Author and Book entities:

Author entity

```
@Entity
public class Author {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @OneToMany(mappedBy = "author")
    private List<Book> books;

    // Getters and setters
}
```

Book entity

```
@Entity
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String title;

    @ManyToOne
    @JoinColumn(name = "author_id")
    private Author author;

    // Getters and setters
}
```


Here, the **mappedBy** attribute in the **@OneToMany** annotation of the **Author** entity refers to the **author** field in the **Book** entity. This indicates that the **Book** entity's **author** field maps the relationship, meaning that the **Author** entity is not the owner of the relationship in the database schema.

In summary, **@JoinColumn** is used to define the column that maintains the relationship in the database. In contrast, **mappedBy** is used to indicate the field in the owning entity that manages the bidirectional relationship. These annotations help Hibernate understand how the entities are related and how they should be mapped to the database tables.

References:

1. [JPA Entites](#)
2. [JPA Unique Constraints](#)
3. [Join Column](#)
4. [Jpa JoinColumn VS mappedby](#)
5. [JPA and Hibernate](#)

Spring Data and JPA

Introduction to Spring Data and JPA

With Spring Data JPA, developers can write data access code using JPA and benefit from Spring Data's repository abstraction and other features. Spring Data JPA provides a set of base repository interfaces, such as `JpaRepository` and `CrudRepository`, which offer a range of methods for performing common data access operations, such as querying, saving, and deleting entities.

Spring Data JPA also supports query creation from method names, allowing developers to write queries using a simple method naming convention rather than complex JPQL or SQL queries. Additionally, Spring Data JPA supports pagination, sorting, and auditing, among other features.

Overall, Spring Data and JPA combine to provide a powerful and flexible approach to data access in Spring applications, allowing developers to write concise, efficient, and portable data access codes.

JDBC vs Hibernate vs Spring Data vs Spring Data JPA

JDBC, Hibernate, Spring Data, and Spring Data JPA are all technologies used in Java development for data access. Here are the differences between them:

- JDBC (Java Database Connectivity) is a low-level API for accessing relational databases in Java. It provides standard classes and interfaces for connecting to a database, executing SQL statements, and managing database transactions. Developers must write raw SQL statements and manually handle exceptions and database connections.
- Hibernate is a popular object-relational mapping (ORM) framework that provides a higher-level abstraction for data access than JDBC. It maps Java objects to database tables and provides a set of annotations and APIs for defining the mapping. Hibernate supports many relational databases and provides a range of features, such as lazy loading, caching, and optimistic locking.
- Spring Data is a higher-level abstraction for data access that provides a consistent programming model for different data stores, such as relational databases, NoSQL databases, and cloud-based data stores. It offers a base repository interface set that provides generic CRUD operations and query methods translated to

database-specific queries at runtime. Spring Data also supports pagination, sorting, auditing, and other features.

- Spring Data JPA is a Spring Data module that integrates with the Java Persistence API (JPA). JPA is a standard API for ORM in Java and provides a set of annotations and interfaces for mapping Java objects to relational databases. Spring Data JPA offers a set of base repository interfaces that extend the JPA repository interfaces and provide additional features, such as query method generation from method names and support for pagination, sorting, and auditing.

In short, JDBC is a low-level API, Hibernate is an ORM framework, Spring Data is a higher-level abstraction for data access, and Spring Data JPA is a module of Spring Data that provides integration with JPA for ORM. Each technology offers a different abstraction and functionality for data access in Java applications.

Spring Data and JPA

Spring Data JPA is a framework that simplifies the development of data access layers in Java applications. It provides a way to interact with databases using object-oriented techniques and reduces boilerplate code.

1. Repository: The Repository interface in Spring Data JPA provides a set of methods to perform CRUD (Create, Read, Update, and Delete) operations on the database. It acts as a mediator between the application and the data source. You can define custom methods in your repository interface to perform complex queries or operations.

2. CRUD Repository: The CrudRepository interface extends the Repository interface and provides additional methods to perform CRUD operations. It has predefined methods like `save()`, `findById()`, `findAll()`, `deleteById()`, etc.

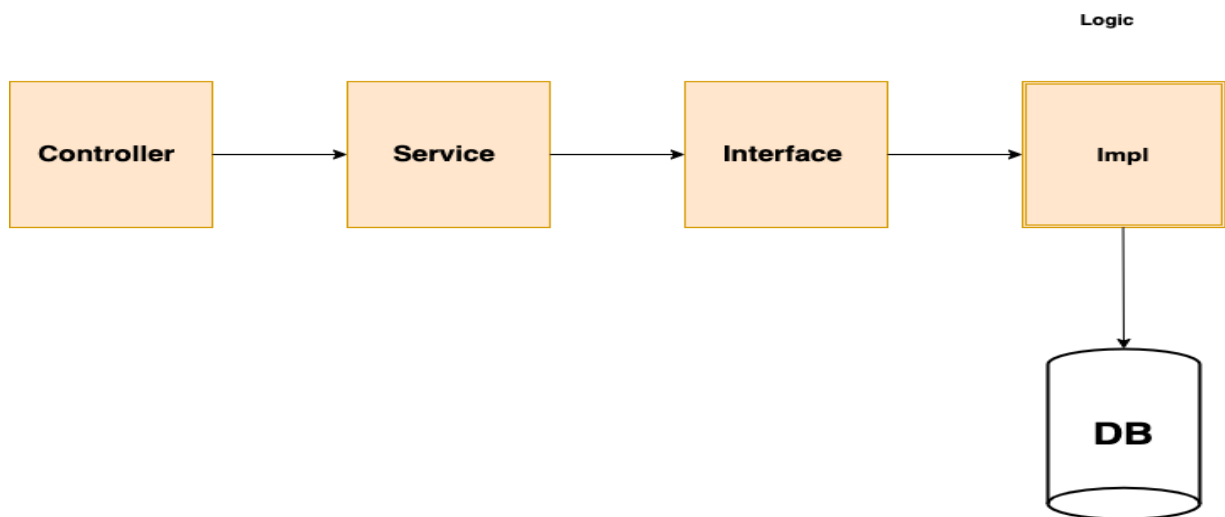
3. Paging and Sorting: The PagingAndSortingRepository interface extends the CrudRepository interface and provides methods for pagination and sorting. It has methods like `findAll(Pageable pageable)` and `findAll(Sort sort)` to get data with pagination and sorting.

4. JPA Repository: The JpaRepository interface extends the PagingAndSortingRepository interface and provides additional methods to work with JPA (Java Persistence API). It has methods like `flush()`, `deleteInBatch()`, `saveAndFlush()`, etc., that work with JPA's EntityManager.

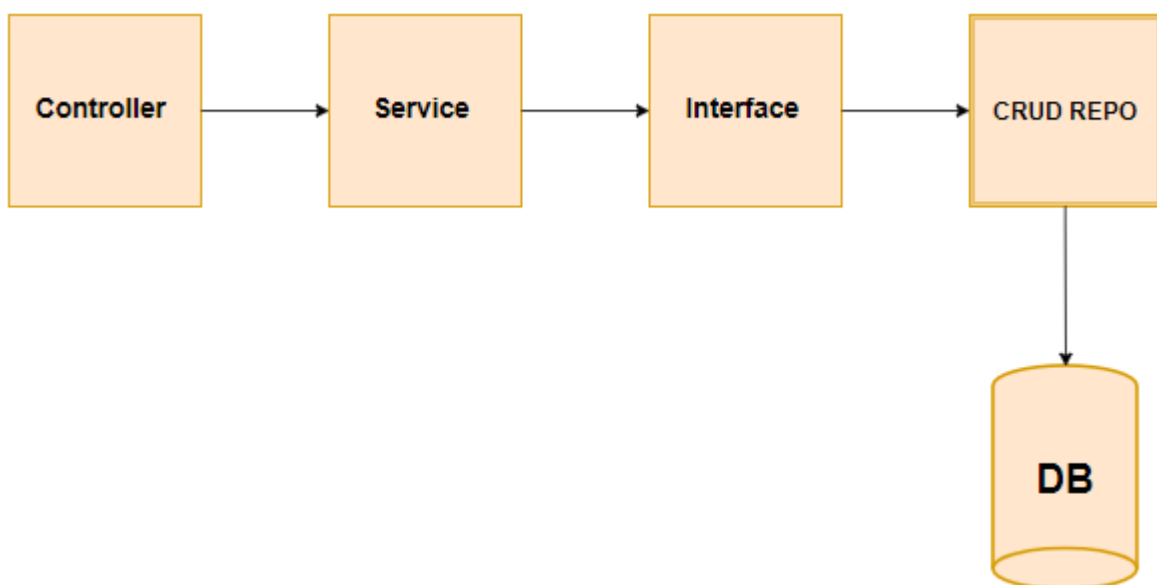
Overall, Spring Data JPA provides much functionality to simplify database access in Java applications. The repository interfaces provide a consistent way to interact with the database and reduce boilerplate code. The paging and sorting support makes it easy to work with large data sets, and the JPA repository provides additional functionality for working with JPA.

Understanding the flow

Earlier



Now



Implementing the API

Now we are required to make some changes in the previous project to use Crud Repository

itemRepository

```
package com.cn.cnkart.dal;

import org.springframework.data.repository.CrudRepository;

import com.cn.cnkart.entity.Item;

public interface ItemRepository extends CrudRepository<Item, Integer> {

}
```

itemDetailsRepository

```
package com.cn.cnkart.dal;

import org.springframework.data.repository.CrudRepository;

import com.cn.cnkart.entity.ItemDetails;

public interface ItemDetailsRepository extends CrudRepository<ItemDetails,
Integer>{

}
```

Application.yml file

```
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/CNkart
    username: root
    password: password
    driver-class-name: com.mysql.cj.jdbc.Driver
  jpa:
    hibernate.ddl-auto: update
```

itemService

```
package com.cn.cnkart.service;

import java.util.ArrayList;
import java.util.List;

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

import com.cn.cnkart.dal.ItemRepository;
import com.cn.cnkart.entity.Item;

@Service
public class ItemService {

    @Autowired
    ItemRepository itemRepository;

    public Item getItemById(int id) {
        return itemRepository.findById(id).get();
    }

    public void saveItem(Item item) {
        itemRepository.save(item);
    }

    public void delete(int id) {
        itemRepository.deleteById(id);
    }

    public void update(Item updateItem) {
        itemRepository.save(updateItem);
    }

    public List<Item> getItem() {
        List<Item> itemList = new ArrayList<>();
        itemRepository.findAll().forEach(item -> itemList.add(item));
        return itemList;
    }
}
```

itemDetail service

```
package com.cn.cnkart.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.cn.cnkart.dal.ItemDetailsRepository;

@Service
public class ItemDetailsService {

    @Autowired
    ItemDetailsRepository itemDetailsRepository;

    public void delete(int id) {
        itemDetailsRepository.deleteById(id);
    }
}
```

Conclusion

In conclusion, Spring Data JPA is a powerful tool for Java developers who need to interact with relational databases. It simplifies creating and executing database queries and provides several features that make working with databases more efficient and effective.

By using Spring Data JPA, developers can write less code, reduce the amount of boilerplate required to interact with databases and take advantage of the advanced features provided by the framework, such as caching, lazy loading, and auditing.

Overall, Spring Data JPA is a valuable addition to the Spring ecosystem that can help developers build robust and scalable applications more quickly and easily.

Instructor Codes

- [CNKart Application](#)

References

1. [JDBC vs Hibernate vs JPA vs Spring Data JPA](#)
2. [Spring Data JPA Official Documentation](#)
3. [CRUDRepository example](#)

SQL Cheat Sheet

This cheat Sheet is designed to assist learners in understanding some of the basic SQL queries. This would be helpful when learning database operations in Spring Boot with suitable examples. Throughout this sheet, we will use a scenario to see how to employ different types of SQL commands.

Scenario: You have a database named **"Company"** with two tables: **"Employees"** and **"Departments"**. The **"Employees"** table contains employee information, such as their ID, name, department ID, and salary. The **"Departments"** table contains department information, such as their ID and name.

id	name
1	Sales
2	Finance

Department Table

id	name	salary	department_id
1	Ramesh Verma	20000	2
2	Mahesh Anand	25000	1

Employee Table

1. SELECT statement:

- Retrieve all columns from the Employee table:

```
SELECT * FROM Employees;
```

- Retrieve specific columns from the Employee table:

```
SELECT name, salary from Employees;
```


- c. Select unique salaries from the Employee table:

```
SELECT DISTINCT salary from Employees;
```

- d. Select salary as alias compensation:

```
SELECT salary as Compensation from Employees;
```

- e. Filter rows based on a condition:

```
SELECT * from Employees WHERE condition;
```

- f. Sort the employees based on their salaries:

```
SELECT * from Employees ORDER BY SALARY ASC|DESC;
```

- g. Group rows based on a column and apply aggregate functions:

```
SELECT column, function(column) FROM table_name GROUP BY column;
```

The aggregate function can be min(), max(), avg() etc. We will discuss these functions below.

- h. Limit the number of rows returned in the result set:

```
SELECT name, salary from Employees LIMIT 1000;
```

2. Matching:

- a. Matching data using **LIKE**

```
SELECT * from Employees WHERE name="J%";
```

This query returns the list of employees whose name starts with J.

- b. We can also use regular expressions to match data.

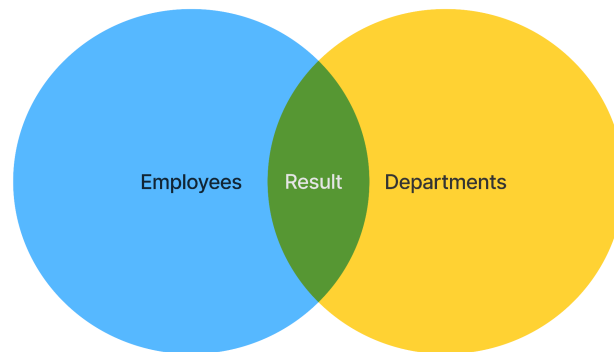
```
SELECT * from Employees WHERE name="regex";
```

3. Joins:

- a. **Inner Join:** Retrieve employee name and department name for employees in the "IT" department:

```
SELECT Employees.name, Departments.name  
FROM Employees  
JOIN Departments ON Employees.department_id = Departments.id  
WHERE Departments.name = 'IT';
```

The inner join returns rows with a match on both tables.



- b. **Outer Join:** The outer join returns all rows from the left table (table1) and the right table (table2). If there is no match, NULL values are returned for the columns of the respective non-matching table.

Example: Retrieve employee name and department name for all employees and departments, including unmatched records from both tables.

```
SELECT Employees.name, Departments.name
FROM Employees
FULL OUTER JOIN Departments
ON Employees.department_id = Departments.id;
```

- c. **Left Join:** The left join returns all rows from the left table (table1) and the matched rows from the right table (table2). If there is no match, NULL values are returned for the columns of the right table.

Example: Retrieve employee name and department name for all employees, including those without a department assigned.

```
SELECT Employees.name, Departments.name
FROM Employees
LEFT JOIN Departments
ON Employees.department_id = Departments.id;
```

- d. **Right Join:** The right join returns all rows from the right table (table 2) and the matched rows from the left table (table 1). If there is no match, NULL values are returned for the columns of the left table.

Example: Retrieve employee and department names for all departments, including those without employees.

```
SELECT Employees.name, Departments.name
```

```
FROM Employees  
RIGHT JOIN Departments  
ON Employees.department_id = Departments.id;
```

4. MYSQL Calculation Functions:

a. Perform Mathematical Calculations:

Example: Calculate the total salary for an employee, including a 10% bonus:

```
SELECT salary, salary * 1.1 AS total_salary FROM Employees;
```

b. Get the maximum salary in the company:

```
SELECT Max(salary) AS maximum_salary FROM Employees;
```

c. Get the minimum salary in the company:

```
SELECT MIN(salary) AS minimum_salary FROM Employees;
```

d. Calculate the sum of salaries of all employees in the company:

```
SELECT SUM(salary) AS sum_of_salaries FROM Employees;
```

e. Calculate the number of employees in the company

```
SELECT COUNT(*) AS number_of_employees FROM Employees;
```

5. String functions in MYSQL

a. **SUBSTR:** The SUBSTR function extracts a substring from a string based on a specified starting position and length.

Example: Extract the first three characters of an employee's name.

```
SELECT SUBSTR(name, 1, 3) AS initials FROM Employees;
```

b. **UPPER:** The UPPER function converts a string to uppercase.

Example: Convert the employee names to uppercase.

```
SELECT UPPER(name) AS uppercase_name FROM Employees;
```

c. **LOWER:** The LOWER function converts a string to lowercase.

Example: Convert the department names to lowercase.

```
SELECT LOWER(name) AS lowercase_name FROM Departments;
```

- d. **STRCMP:** The STRCMP function compares two strings and returns 0 if they are equal, a negative value if the first string is less than the second, or a positive value if the first string is greater than the second.

Example: Compare the names of two employees.

```
SELECT STRCMP(name, 'John Doe') AS name_comparison FROM Employees;
```

- e. **CONCAT:** The CONCAT function concatenates two or more strings together.

Example: We have first and last names in different columns and want to concatenate them.

```
SELECT CONCAT(first_name, ' ', last_name) AS full_name FROM Employees;
```

Derived Queries

A derived query in Spring Data JPA refers to a query method automatically generated by the framework based on the method name. It allows developers to define data retrieval operations concisely and readably without writing explicit JPQL (Java Persistence Query Language) or SQL queries. Derived queries are especially useful for common and straightforward operations.

Here are the critical aspects of derived queries in detail:

1. Method Naming Conventions: Derived queries are created by following a specific naming convention in the repository interface. The method name consists of a combination of keywords that convey the intended action and the properties of the entity class involved.

2. Keywords: Keywords such as *findBy*, *findAllBy*, *readBy*, *getBy*, and logical operators like **And**, **Or**, and **Not** are used to construct the method names. These keywords indicate the operation and the criteria to be applied to the query.

3. Property Expressions: After the keywords, the properties of the entity class are referred to using their camel-case names. For example, if you have an Employee entity with a property named `firstName`, the method `findByFirstName` would be generated.

4. Comparison Keywords: Comparison keywords like `Equals`, `IsNull`, `Like`, `GreaterThan`, `LessThan`, and more can be used to specify conditions for filtering data.

5. Logical Operators: Logical operators like `And` and `Or` allow you to combine conditions within the query method, providing more advanced filtering capabilities.

6. Ordering and Limiting: Derived queries can include ordering (using `OrderBy`) and limiting the number of results (using `First`, `Top`, `Distinct`, and others).

7. Negation: The `Not` keyword allows the negation of a condition. For instance, `findByActiveIsNotTrue` fetches records where the active property is not true.

8. Multiple Parameters: Derived queries can accept multiple parameters, allowing you to specify various criteria for filtering data.

9. Null Handling: Derived queries handle null values automatically. For example, `findByEmailIsNull` retrieves records where the email is null.

10. Derived Query Limitations: While derived queries are powerful for simple and common queries, they might not cover complex scenarios that require dynamic conditions, joins, subqueries, or custom calculations. You can use `@Query` annotations with custom JPQL or SQL queries in such cases.

Let's see derived query examples with the help of a scenario.

Scenario: You have a **"Company"** database with a table: **"Employee"**. The **"Employee"** table contains employee information, such as their ID, first name, last name, salary, age etc.

id	first_name	last_name	age	email	salary
1	Ramesh	Verma	25	rv@gmail.com	20000
2	Mahesh	Anand	26	ma@gmail.com	25000

Employee Table

1. To retrieve all employees with the first name **"John."**

```
List<Employee> findByFirstName("John");
```

2. To retrieve employees with a last name "Doe" and age greater than or equal to 30:

```
List<Employee> findByAgeGreaterThanOrEqualToAndLastName(30, "Doe");
```

3. To retrieve employees whose email contains "example.com":

```
List<Employee> findByEmailLike("%yahoo.com%");
```

4. To retrieve employees with a salary between 40000 and 60000:

```
List<Employee> findBySalaryBetween(40000, 60000);
```

5. To retrieve all employees ordered by their joining date in descending order:

```
List<Employee> findAllOrderBySalaryDateDesc();
```

6. To retrieve employees with a salary greater than a certain amount, ordered by salary in ascending order:

```
List<Employee> findBySalaryGreaterThanOrderBySalaryAsc(double salary);
```

Spring Data JPA Queries

Introduction

Spring Data JPA is a sub-project of Spring Data and provides an abstraction over the Data Access Layer using Java Persistence API, simplifying the interaction with databases in a Spring Boot application. It provides convenient ways to query and manipulate data using predefined query methods, derived queries, JPQL (Java Persistence Query Language), and native queries. Let's explore the different types of queries in Spring Data JPA:

Derived Queries

Derived queries in Spring Boot provide a convenient way to define queries in your repositories by deriving the query from the method name. This feature is made possible by Spring Data JPA, a robust abstraction layer built on top of JPA (Java Persistence API), makes this feature possible. Here's how you can leverage derived queries in Spring Boot:

1. Repository Definition

- a. Start by defining an interface for your repository. Typically, you extend either the `CrudRepository` or `JpaRepository` interface, which provides basic CRUD (Create, Read, Update, Delete) operations.

For Example,

```
public interface ItemDetailsRepository extends JpaRepository<ItemDetails, Integer>
{
    // Code to be written
}
```

In this example, `ItemDetailsRepository` extends `JpaRepository` and operates on the `ItemDetails` entity with a primary key of type integer.

2. Method Naming Convention

- a. Derived queries rely on a naming convention to generate the SQL query. The method name should follow a specific pattern, where the query criteria are expressed using the method name keywords.
- b. The general structure of a derived query method is *findXByY*, where **X** represents the type of object you want to retrieve, and **Y** denotes the property or field on which the query will be based.
- c. You can also include additional keywords to express specific conditions or ordering requirements. Some commonly used keywords include And, Or, Between, LessThan, GreaterThan, OrderBy, and more.
- d. Here are a few examples of derived queries:

```
public interface ItemDetailsRepository extends JpaRepository<ItemDetails, Integer>
{
    List<ItemDetails> findByPriceGreaterThan(double price);
    List<ItemDetails> findByDescriptionOrderByPriceDesc(String description);
    List<ItemDetails> findByCategoryAndBrand(String Category, String brand);
}
```

- e. In the above examples, the queries will be generated based on the method names. For instance, `findByDescriptionOrderByDesc` will retrieve `ItemDetails` whose description matches the specified value and order them in descending order. `Findbycategoryandbrand` will retrieve `ItemDetails` whose `Category` and `Brand` match the specified values.

3. Execution and Results

- a. When you invoke a derived query method, Spring Data JPA automatically generates the corresponding SQL query based on the method name and executes it on your behalf.
- b. The return type of the query method should match the type you want to retrieve. In the examples above, we used `List<ItemDetails>`, so the result will be a list of `ItemDetails` objects.
- c. You can include method parameters in the query method to filter the results further. Spring Data JPA will automatically map these parameters to the corresponding query conditions.
For example: The query below will fetch `itemDetails` with the given description and category.

```
List<ItemDetails> findByDescriptionAndCategory(String description, String category);
```


- d. You can also use other keywords like Or to combine conditions, Between to specify a range, and more to create complex queries.

Derived queries offer a concise and expressive way to define queries in Spring Boot applications, reducing the need to write explicit SQL statements. Following the naming convention and leveraging method parameters, you can quickly build queries for various conditions and retrieve the desired data from your database.

JPQL

JPQL (*Java Persistence Query Language*) is a query language specifically designed for working with entity objects in Java-based applications using the Java Persistence API (JPA). It is an object-oriented query language that allows you to perform database queries and manipulations in a database-agnostic way, i.e. it is independent of the type of database used. Here's a detailed description of JPQL and its key features:

1. Object-Orientation

- a. JPQL operates on entity objects and their relationships rather than directly dealing with database tables and columns.
- b. It allows you to write queries using your Java code's entity class names, properties, and relationships.

2. Syntax and Structure

- a. JPQL queries resemble SQL queries, but they are written more object-oriented.
- b. The basic structure of a JPQL query consists of a SELECT clause, an optional FROM clause, and optional WHERE, GROUP BY, HAVING, and ORDER BY clauses.
- c. Here's an example of a simple JPQL query:
In the query below, Employee is the entity class name, e is an alias for the Employee entity, and the department is a named parameter.

```
SELECT e FROM Employee e WHERE e.department = :department
```

3. Entity and Property Names

- a. JPQL uses entity class names and their properties to reference database tables and columns.
- b. Entity class names are case-sensitive and should match the exact name of the entity class.
- c. Property names in JPQL correspond to the fields or properties defined in the entity class.

4. Named Parameters

- a. JPQL supports named parameters that start with a colon (:) followed by a parameter name.
- b. Named parameters to pass dynamic values into queries, such as filtering criteria.
- c. Parameters can be used in the WHERE clause, ORDER BY clause, and other query parts.
- d. Here is an example of a JPQL query:

```
@Query("Select itd from ItemDetails itd where itd.category=?1  ORDEY BY  
itd.price DESC")  
List<ItemDetails> findByCategoryOrderByPrice(String category);
```

5. Querying Entities

- a. JPQL allows you to retrieve entities based on various criteria using the SELECT and FROM clauses.
- b. JPQL allows you to retrieve entities based on various criteria using the SELECT and FROM clauses.

6. Querying Relationships

- a. JPQL supports navigating and querying relationships between entities.
- b. You can traverse relationships using dot notation, such as e.department.name, to access properties of related entities.
- c. Joins can be performed using the JOIN keyword to fetch related entities or perform complex queries involving multiple entities.

7. Aggregation and Grouping

- a. Using the GROUP BY and HAVING clauses, JPQL supports aggregations such as COUNT, SUM, AVG, MIN, and MAX.
- b. You can group entities based on specific properties and apply aggregate functions to those groups.

8. Ordering Results

- a. JPQL supports ordering query results using the ORDER BY clause.
- b. You can specify one or multiple properties by which the results should be sorted, along with ascending or descending order.

9. Integration with JPA

- a. JPQL is tightly integrated with the Java Persistence API (JPA) and can be used with JPA providers such as Hibernate, EclipseLink, and OpenJPA.
- b. JPA providers translate JPQL queries into the corresponding SQL statements specific to the underlying database.

JPQL provides a powerful and flexible way to query and manipulate entity objects in a database-agnostic manner. It allows you to express complex queries using familiar Java entity classes and their relationships. JPQL queries can be dynamically constructed and executed within JPA-based applications, making it a fundamental tool for database interactions in Java.

Native Queries

Native queries in the context of Java Persistence API (JPA) allow you to execute SQL queries directly against the underlying database. Unlike JPQL (Java Persistence Query Language), which is a database-agnostic query language, native queries provide a way to work with database-specific SQL statements. This feature is useful when leveraging database-specific features or optimising performance for complex queries.

Here's a detailed description of native queries and their key features:

1. SQL-Based Queries

- a. Native queries are written in SQL (Structured Query Language), the standard language for interacting with relational databases.
- b. You can write SQL statements directly as strings to execute queries against the database.
- c. SQL queries are specific to the database vendor and may not be portable across different systems.
- d. Here is an example of SQL-based Native query,

```
@Query(value = "Select * from item where description like :desc LIMIT 4",  
nativeQuery = true)  
List<Item> getItemByDesc(@Param("desc") String desc);
```

2. EntityManager Execution

- a. In JPA, native queries are executed using the EntityManager interface, representing the persistence context and allowing interaction with the database.

- b. The `createNativeQuery` method of `EntityManager` is used to create a native query object.
- c. You can then set parameters, execute the query, and retrieve the results using methods provided by the native query object.

3. Parameter Binding

- a. Native queries support parameter binding to safely pass dynamic values to the query.
- b. Parameters can be positional or named. Positional parameters are denoted using question marks (?), while named parameters are prefixed with a colon (:).
- c. Parameter values can be set using the `setParameter` or `setParameterByName` methods of the native query object.

4. Result Mapping

- a. Native queries return results as database-specific result sets.
- b. By default, the result set is returned as an array of objects, with each element representing a column value.
- c. You can also map the result set to specific entity classes or use column aliases to map selected columns to specific properties.

5. Scalar Results

- a. Native queries can retrieve scalar values, such as integers or strings, using aggregate functions or selecting specific columns.
- b. Scalar results are returned as individual or arrays of values, depending on the query and mapping configuration.

6. Transaction Management

- a. Native queries operate within the context of a JPA transaction.
- b. If a transaction is inactive, JPA automatically starts a new transaction before executing the native query and commits or rolls back the transaction accordingly.

Native queries provide flexibility and direct access to the database, allowing you to leverage database-specific features and optimise performance when needed. However, it's important to note that native queries tie your code to a specific database, reducing portability across different database systems. Therefore, it's recommended to use native queries sparingly and favour JPQL for most database operations in JPA-based applications, as JPQL offers a more database-agnostic approach.

Named and NamedNative Queries

Named and named native queries in JPA provide a way to define and reuse queries using a unique name instead of writing the query directly in code. Both named and native queries offer improved code readability, query management, and potential performance optimisations. Let's explore them in detail:

Named Queries

1. Named queries are SQL-like queries written in JPQL (Java Persistence Query Language) and are database-agnostic.
2. They are defined and associated with an entity class using annotations or XML configuration files.
3. Named queries provide a level of abstraction over the underlying SQL statements and can be used across different database systems.
4. Benefits of named queries include improved maintainability, reusability, and the ability to externalise query definitions.
5. Here is an example of NamedQuery:

```
@Entity
@NamedQuery(name = "ItemDetails.findByCategoryOrderByPrice",
    query = "Select itd from ItemDetails itd where itd.category=?1 ORDEY BY
    itd.price DESC")
@Table(name = "item_details")
public class ItemDetails {
    //...
}

public class ItemDetailsRepository extends JpaRepository<ItemDetails,
Integer> {
    List<ItemDetails> findByCategoryOrderByPrice(String category);
}
```

NamedNative Queries

1. Named native queries allow you to define and use database-specific SQL queries directly, leveraging the power of native SQL.
2. They are defined and associated with an entity class using annotations or XML configuration files.
3. Named native queries are useful when you need to utilise database-specific features or optimise performance with complex SQL statements.
4. However, they tie your code to a specific database system and are not portable across different systems.
5. Here is an example of NamedNative Query

```
@Entity
@NamedNativeQuery(name = "Item.getItemByDesc"
query = "Select * from item where description like CONCAT(?1, '%') LIMIT
4",
resultClass = Item.class)
@Table(name = "item_details")
public class Item {
    //...
}

public class ItemRepository extends JpaRepository<ItemDetails, Integer>
{
    @Query(name = "Item.getItemByDesc", nativeQuery=true)
    List<Item> getItemByDesc(String desc);
}
```

Named queries and named native queries provide a way to define queries separately from the code logic, offering better query management and code organisation. They can be easily reused, shared across multiple entities, and optimised by the JPA provider. However, it's important to note that named queries should be used judiciously, ensuring they align with your application's requirements and adhere to best practices.

Conclusion

In conclusion, Spring Data JPA provides multiple options for querying and interacting with the database in a Spring Boot application. The choice of query approach depends on the complexity of the query, the need for database-specific features, and the level of control and flexibility required.

- **Derived queries** offer a convenient and concise way to define queries based on method names in the repository interface. They are automatically generated by Spring Data JPA based on the method names and parameter names, reducing the need for boilerplate code. Derived queries are suitable for simple queries and are easily maintained and understood.
- **JPQL (Java Persistence Query Language)** is a database-agnostic query language that allows you to write queries using entity classes, properties, and relationships. It provides a more expressive and powerful way to construct queries with support for filtering, sorting, and aggregations. JPQL is suitable for more complex queries and offers a good balance between readability and flexibility.
- **Native queries** enable you to write SQL queries directly and leverage database-specific features and optimisations. They provide the most control and flexibility over the query structure, allowing you to write complex queries and take advantage of advanced database capabilities. Native queries are helpful for scenarios requiring fine-tuned performance, database-specific functionalities, or complex data transformations.

When choosing the appropriate query approach, consider the trade-offs between simplicity, maintainability, portability, and performance. Derived queries are a good choice for common use cases, providing a simple and intuitive way to define queries. JPQL offers a more expressive and database-agnostic approach, balancing convenience and flexibility. Native queries should be used judiciously, primarily for advanced scenarios requiring fine-grained control over the SQL statements and database-specific features.

Spring Data JPA simplifies database operations, promotes code reuse, and improves productivity in developing data access layers in Spring Boot applications.

Instructor Codes

- [CNKart Application](#)

References

1. [Official Documentation](#)
2. [Derived Queries](#)
3. [Derived Queries II](#)
4. [JPLQ](#)
5. [Joins using JPA](#)
6. [Native Queries](#)
7. [NamedNative Queries](#)
8. [Query Examples](#)