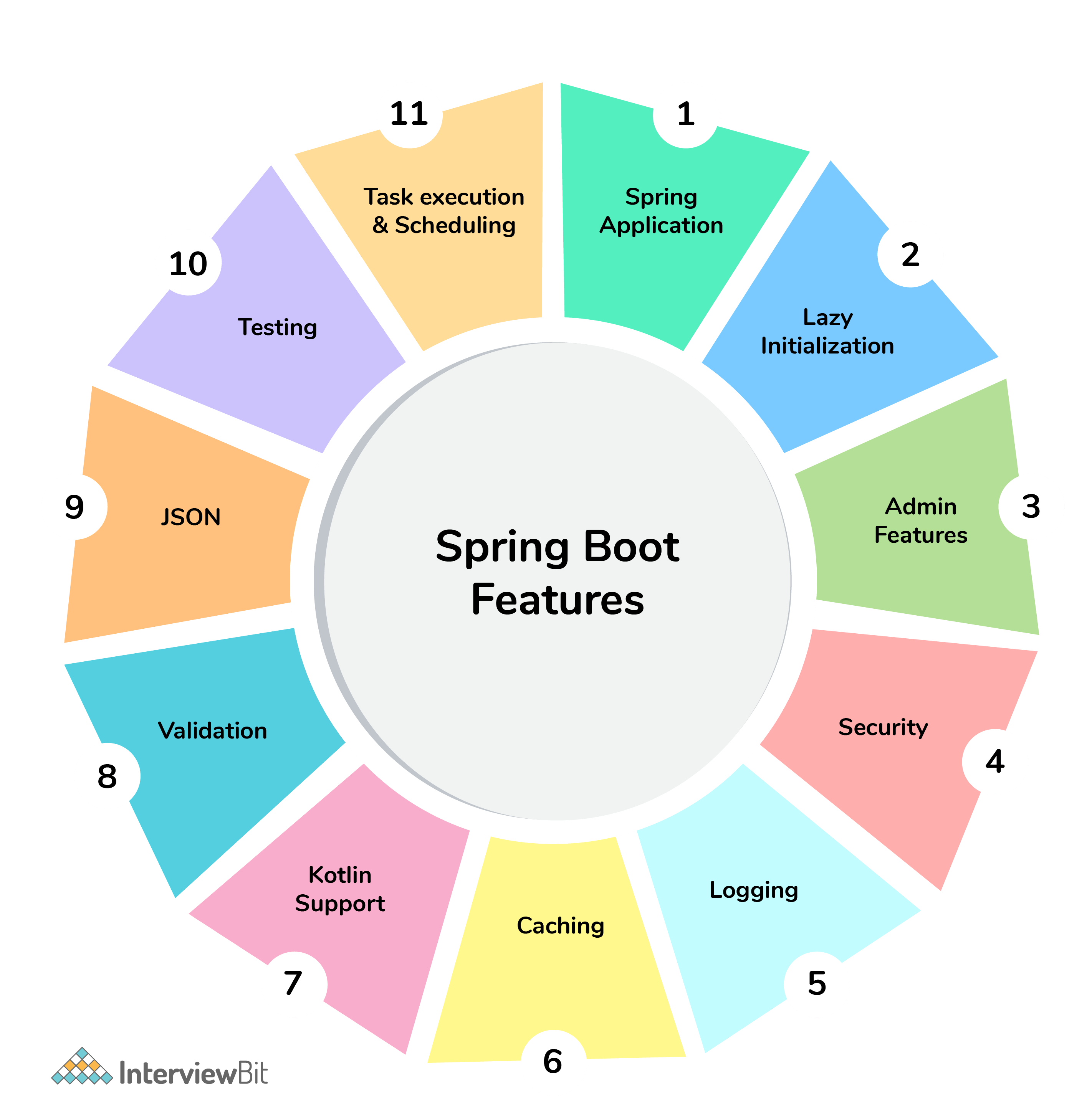
**Spring Boot**

**Que: What is Spring Boot?  
Ans:** Spring Boot is a Java framework that makes it easier to create and run Java Applications.

* It simplifies the configuration and setup process, allowing developers to focus more on writing code and business logic for their application.
* Spring boot is the module of spring framework, facilitate Rapid Application Development(RAD) Capabilities.
* Spring boots solve many developers problem:
* Auto Configuration
* Dependency Management
* Embedded Server- Tomcat]



**Que: Why Spring Boot over Spring? (Spring Boot Added Feature)**

**Ans:** It’s not an alternative to spring. Spring Boot uses Spring behind the scenes. Spring boot provides many advantages over normal spring framework.

a. Easy to use- Remove boilerplate codes.

b. Production Ready Applications: Metrix, Health Check and many features are designed for the production ready applications.

c. Rapid Development- Opinionated approach and auto-configuration enable developers to quickly develop apps.

d. Provides Dependency Managements: no need to manage separate dependencies manually.

e. Autoconfiguration: Spring Autoconfigures thing by defaults.

F. Embedded Severs: Tomcat.

g. In memory db

h. Version Management

i. Actuators

**Que: Why to upgrade from Legacy to Spring Boot? (Spring Boot Added Feature)**

**Ans:** Upgrading to Spring Boot from a legacy Spring application offers several benefits and advantages, including:

* **Simplified Configuration**: Spring Boot eliminates the need for XML-based configuration by providing sensible defaults and auto-configuration.
* **Reduces boilerplate code**.
* **Rapid Development:** Spring Boot enables rapid development by providing a set of pre-configured starters and a streamlined development experience.
* **Embedded Servers:** Spring Boot includes embedded web servers (such as Tomcat, Jetty, or Undertow) by default.
* **Dependency Management:** Spring Boot simplifies dependency management by providing dependency management tools and predefined dependency configurations.
* **Microservices Architecture Support**: Spring Boot is well-suited for building microservices-based architectures.
* **Actuator Endpoints**: Spring Boot Actuator provides built-in monitoring and management endpoints for monitoring application health, metrics, and diagnostics.
* **Cloud-Native Features**: Spring Boot offers features and integrations for building cloud-native applications.
* **Security Enhancements**: Spring Boot includes security features and integrations out-of-the-box, such as Spring Security, which simplifies the implementation of authentication and authorization in applications.

**Que: How does Spring Boot Works?**

**Ans:**

* Spring Boot automatically configures your application based on the dependencies you have added to the project by using annotation.
* The entry point of the spring boot application is the class that contains @SpringBootApplication annotation and the main method.
* Spring Boot automatically scans all the components included in the project by using @ComponentScan annotation.

**Que: Working of Spring Boot?**

**Ans:**

1. Spring Boot starts by scanning the starter dependencies in pom.xml.

Then download and autoconfigure the module as you include in pom.xml.

Download will only done when is click on Maven Built.

Auto-configure will be done once you run the code.

2. For Example we have to create web application the we have to put

**spring-boot-starter-web** dependency in pom.xml.

when we start the project spring boot download all the dependency required for web and configure the things like spring mvc.

**Que: How spring boot starts?**

**Ans:**

1. Starts by calling main() method of you main class.

**SpringBootApplication.**



2. The run () method of SpringApplication is called. This method starts the application by creating an application context and initializing it.

(Application context is spring contains jisme sare beans hai).

3. @SpringBootApplication triggers.

4. Once the application context is initialized, the run () method starts the application’s embedded web server.

**Ques: What are top Spring Boot Annotations?**

**Ans:**

* **@SpringBootApplication**: It is present in the main class of the application. Its combination of three annotations:

**@Configuration:** Indicates that the class declares one or more @Bean methods and can be processed by the Spring container to generate bean definitions and service requests for those beans at runtime.

🡪jitne bhi beans hai wo application context me register ho jayegi configure ho jayegi.

**@EnableAutoConfiguration**: Enables Spring Boot's auto-configuration mechanism that automatically configures the Spring application based on the dependencies and customizations present in the classpath.

**@ComponentScan:** Enables component scanning for Spring components (such as @Controller, @Service, @Repository, @Component, etc.) so that these components can be automatically discovered and registered with the Spring application context.

* **@Component:** It is used to mark a class as a Spring Bean that will be managed by Spring Container. @Component considers a class is a Spring-managed component and should be automatically detected and registered during component scanning.

🡪Kisi bhi class ko as a spring bean create kr skate hai.

Making it available for dependency injection or other Spring-managed activities.

When Spring scans your application for components (usually at startup), it identifies classes annotated with @**Component** (or its specializations such as @**Service**, @**Repository**, and **@Controller**, which are themselves meta-annotated with @**Component**) and registers them as Spring beans in the application context.

**@Autowired:** This annotation is used to automatically inject dependencies into a spring managed bean.

It helps in achieving the Inversion of Control (IoC) principle by letting Spring resolve and inject collaborating beans into your bean automatically.

* **@Service:** The @Service annotation in the Spring Framework is a specialization of the @Component annotation. It indicates that a particular class is a service class in the business logic layer of an application. This annotation serves primarily to distinguish service classes from other types of components like controllers or repositories.

🡪 Is class me business logic likha hai. koi bhut zaruri ni hai likhna bs best practice hota hai.

* **@RestController:** Mark the class as REST Controller. The @RestController annotation in Spring MVC combines the functionality of @Controller and @ResponseBody.

**@Controller:** In Spring MVC and Spring Boot, the @Controller annotation is used to mark a class as a controller component in the MVC architecture. Controllers in Spring handle incoming HTTP requests, process them, and return an appropriate response.

**@ResponseBody:** In Spring MVC and Spring Boot, the @ResponseBody annotation is used to indicate that the return value of a method should be bound to the web response body.

* **@RequestMapping:** In Spring MVC and Spring Boot, @RequestMapping is a versatile annotation used to map HTTP requests to handler methods in controller classes. It's a fundamental building block for defining the URL patterns that your application will respond to.

Used to map URL to method. Used on class as well as method level.

* **@Repository:** The @Repository annotation in Spring is used to indicate that the annotated class is a repository component. It's typically used to identify classes that are responsible for encapsulating storage, retrieval, and querying of data from a database.

**Que: What are the Spring Boot Starters?**

**Ans**: Spring Boot Starters are a set of dependency descriptors that allow you to quickly and easily add dependencies to your Spring Boot project.

This includes all the dependencies, version control and configurations.

They are essentially pre-configured sets of dependencies that address specific needs, such as web applications, data access, messaging, testing, etc.

* Spring Boot starters follow a naming convention: spring-boot-starter-\*, where \* represents the specific domain or functionality the starter addresses. For example:
* spring-boot-starter-web: Includes dependencies for building web applications using Spring MVC, embedded Tomcat server, and other related libraries.
* spring-boot-starter-data-jpa: Includes dependencies for using Spring Data JPA for data access.
* spring-boot-starter-security: Includes dependencies for integrating Spring Security into your application.
* spring-boot-starter-test: Includes dependencies for testing Spring Boot applications using JUnit, Mockito, and other testing libraries.

To use a Spring Boot starter, you simply need to include it as a dependency in your project's pom.xml (for Maven) or build.gradle (for Gradle) file. Spring Boot's dependency management will handle the rest, resolving and downloading all the required dependencies automatically.

**Que: What is dependency Injection?**

**Ans**: ***Dependency Injection (DI) is a design pattern used in software development to achieve loose coupling between components or classes by externalizing their dependencies****.*

In simple terms, DI is a technique where the dependencies of a class are provided to it from an external source, rather than the class creating or managing its dependencies internally.

In Spring Core, DI is achieved using inversion of control (IOC) containers that manage the dependencies between objects.

**Que: What are Types of DI?**

**Ans:**

**Constructor Injection:**

* The IOC container will inject the dependent bean object into the target bean object by calling the target bean constructor.
* Constructor injection promotes immutability and ensures that all required dependencies are provided when an object is created.

**Setter Injection:**

* The IOC container will inject the dependent bean object into the target bean object by calling the setter method.
* Setter injection allows for optional dependencies and facilitates testing and mocking of dependencies.

**Field Injection:**

* The IOC container will inject the dependent bean object into the target bean object by Reflection API.
* Field injection can lead to tight coupling and is less preferred compared to constructor or setter injection, as it makes dependencies less explicit.

**Que: Difference between Constructor Inject and Setter Injection?**

**Ans:** Constructor is mandatory, and setter is optional.

**Que: What are the benefits of DI?**

**Ans:** Dependency Injection offers several benefits, including:

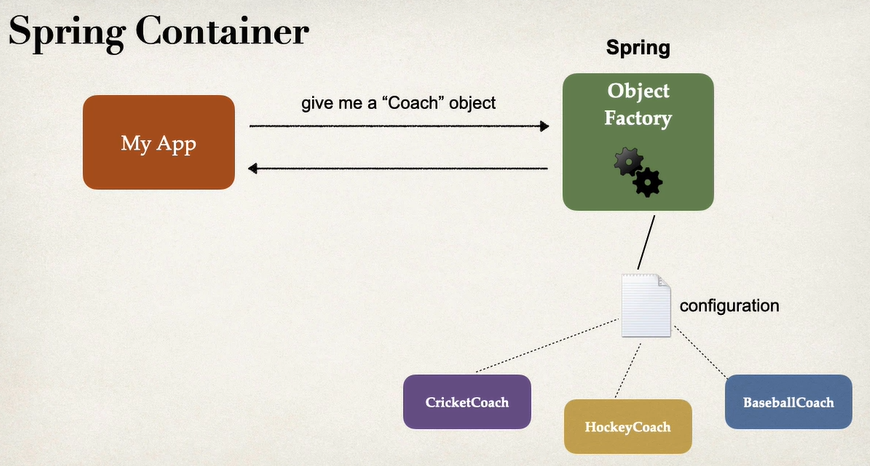
* **Modularity and Encapsulation:** Classes become more modular since they are not responsible for creating their dependencies. This also promotes encapsulation by hiding the creation and management of dependencies from the dependent class.
* **Testability:** DI makes it easier to write unit tests for classes, as dependencies can be easily replaced with mock or stub implementations during testing.
* **Flexibility**: DI allows for easier swapping of dependencies, making the system more flexible and allowing for easier configuration changes.
* **Reduced Coupling:** DI reduces the coupling between classes, making the codebase easier to understand, maintain, and extend.
* In Spring, dependencies can be injected using annotations like @Autowired, XML configuration, or Java configuration.

**Que: What is Inversion of Control?**

**Ans**: ***Inverting the control of creating object using new keyword to container or framework.***

* It the approach of outsourcing the construction and management of objects.
* In traditional programming language, objects were responsible for creating and managing the dependencies and their lifecycle.
* But, In Spring, the control is inverted, and the framework takes charge of managing the dependencies and their lifecycles.
* It allows the user to focus more on business logic rather than managing object creation and configuration.
* In Spring Core, IOC is achieved using IOC containers that manage the lifecycle of objects and their dependencies.
* Ban ko manage karne ka kam spring framework container ko de diya.

***Inversion of Control (IOC) is a design principle that dictates that the control of a program should be handed over to a framework or container, rather than being controlled by the application itself.***

****

**Very Important**

**IOC Container:**

* IOC container is a generic term that refers to a container responsible for managing the lifecycle of objects (beans) and handling their dependencies. It is a design pattern where control of object creation and lifecycle is inverted or delegated to a container rather than being managed by the application code.
* In the context of Spring, the IOC container is implemented through the ApplicationContext interface and its implementations, such as ClassPathXmlApplicationContext or AnnotationConfigApplicationContext.

**Spring Container:**

* The Spring container is an implementation of the IOC container provided by the Spring Framework. It manages the beans, their configurations, dependencies, and lifecycle within a Spring application.
* The Spring container includes various features such as dependency injection, AOP (Aspect-Oriented Programming), event handling, resource management, and more.
* The primary interfaces for the Spring container are ApplicationContext and BeanFactory.

**ApplicationContext:**

* ApplicationContext is a sub-interface of the BeanFactory and represents the Spring container that provides advanced functionalities and features beyond basic bean management.
* ApplicationContext includes capabilities such as event propagation, internationalization (i18n), resource handling, environment profiles, application lifecycle events, and more.
* ApplicationContext is typically used in production-level applications due to its richer feature set compared to BeanFactory.

**BeanFactory**:

* BeanFactory is the root interface for accessing the IOC container in Spring. It provides basic functionalities for managing beans, including bean instantiation, dependency injection, and lifecycle management.
* BeanFactory is the core IOC container interface and is suitable for applications that require lightweight IOC container functionality without the additional features provided by ApplicationContext.
* ApplicationContext is built on top of BeanFactory and extends its functionalities by adding enterprise-specific features.

In summary, IOC container is a generic term for a container managing object lifecycles and dependencies, while the Spring container refers specifically to the IOC container implementation in the Spring Framework. ApplicationContext is an advanced interface that extends BeanFactory and provides additional features such as event handling and internationalization. BeanFactory is the core interface for accessing the IOC container in Spring and provides basic bean management functionalities. The choice between ApplicationContext and BeanFactory depends on the specific requirements and feature needs of the application.

**Que: What is Spring Container?**

**Ans**:

**Spring Container**

The IoC container is responsible to instantiate, configure and assemble the objects.

The IoC container gets information’s from the XML file and works accordingly.

Spring container manages the life cycle of the spring bean (java class) and provides the infrastructure of Dependency injection.

The main tasks performed by IoC container are:

* Create and manage object (Inversion of control)
* Inject object dependencies (Dependency Injection)

**Que: What are the key dependencies in Spring Boot?**

**Ans:**

1. Spring-boot-starter-parent

2. Spring-boot-starter-test

3. Spring-boot-maven-plugin

4. Spring-boot-starter-security

5. Spring-boot-starter-actuator

6. Spring-boot-starter-web

**Que: What is Spring-Boot-Starter-Parent?**

**Ans:**

* spring-boot-starter-parent is a special parent project that Spring Boot provides to simplify the configuration and dependency management of Spring Boot applications.
* It serves as the parent project for your Spring Boot application, allowing you to inherit common configurations, plugin settings, and dependency management from Spring Boot.
* spring-boot-starter-parent is a special parent project that Spring Boot provides to simplify the configuration and dependency management of Spring Boot applications.
* It serves as the parent project for your Spring Boot application, allowing you to inherit common configurations, plugin settings, and dependency management from Spring Boot.

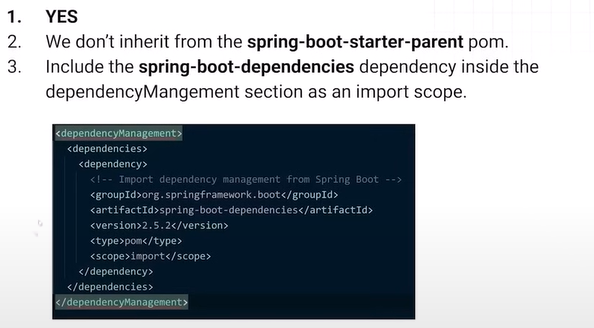
It Helps in:

* Default plugin configurations for Maven or Gradle builds.
* Dependency management, including version management of Spring Boot and related libraries.
* Common properties for Maven or Gradle builds, such as Java version, encoding, and resource filtering settings.
* Common repository definitions for resolving dependencies.



**Que: Can we use Spring Boot Dependency feature and configure maven plugin manually?**

**Ans:**



**Que: What is Spring Boot CLI and What are its Use and benefits?**

**Ans:**

* Spring Boot CLI (Command Line Interface) is a tool provided by the Spring Boot framework that allows you to quickly develop Spring Boot applications from the command line.
* It provides a lightweight way to create, run, and test Spring Boot applications without the need for a full-fledged integrated development environment (IDE).

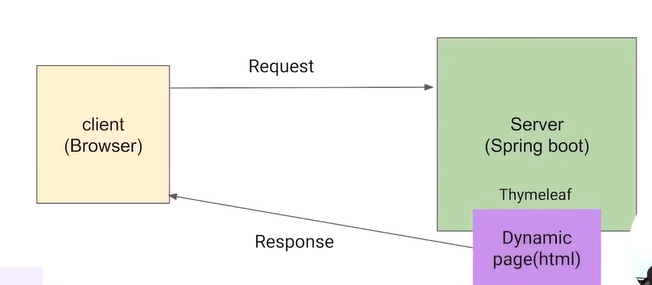
Here are some commonly used commands in the Spring Boot CLI:

* **spring --version**: Displays the version of the Spring Boot CLI.
* **spring init**: Initializes a new Spring Boot project based on the specified options and dependencies.
* **spring run**: Runs a Spring Boot application located in the current directory or specified file.
* **spring jar**: Builds an executable JAR file from a Spring Boot application.
* **spring test:** Executes tests in a Spring Boot application.
* **spring install**: Installs the Spring Boot CLI in your local environment.
* s**pring uninstall**: Uninstalls the Spring Boot CLI from your local environment.

**Que: What is thymeleaf?**

**Ans:**

* It's a popular choice for building web applications in Java and integrates seamlessly with the Spring Framework, although it can be used independently with other frameworks as well.
* Thymeleaf templates are designed to be natural and intuitive, resembling standard HTML, which makes them easy to read and understand even for developers who are not familiar with Thymeleaf.



You can answer I have use Json , Json ko bhej deta hai aur wo frontend par json ko resolve kr ke dikha deta hai.

**Que: Explain Spring Bean Life Cycle?**

**Ans:** Bean: Simple old Java Object.

Class Test{

new student ()🡪ye object is bean

}

**Spring** **Bean**: In spring this bean is manage by IOC Container or spring Container so it is known as Spring bean.

Life Cycle:

In Spring Framework, the lifecycle of a bean refers to the various stages a bean goes through from its instantiation to its destruction.

1.How object is born?

2.How It behaves(whole life)?

3.How it dies?

**Spring Bean life cycle is maintained by IOC Container.**

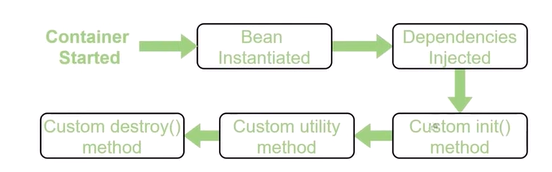
**1. Container starts.**

**2. Container creates object of bean as per request.**

**3. Dependencies is created.**

**4. Dependencies are injected.**

**5. Bean destroyed.**

****

**Custom inti- Initialization:** here you can define custom result, jaise hi bean Banega ye execute ho jayega.

**@PostConstruct:** Annotate a method with @PostConstruct annotation. Spring will invoke this method after the bean has been initialized and its dependencies have been injected.

**@PreDestroy:** The @PreDestroy annotation in Spring is used to indicate a method that should be called just before the bean is destroyed by the Spring container. It's commonly used to perform cleanup or release resources held by the bean before it is removed from the application context.

Note: these both can also be done using overriding i.e., interfaces.

* **InitializingBean Interface:**

The InitializingBean interface defines a single method, afterPropertiesSet(), which is invoked by the Spring container after a bean's properties have been set. This method can be implemented to perform any custom initialization logic required by the bean.

* **DisposableBean Interface**:

The DisposableBean interface defines a single method, destroy(), which is invoked by the Spring container before a bean is destroyed. This method can be implemented to perform any custom cleanup or resource release logic required by the bean.

**Que: What is Bean Factory, What is XMLBeanFactory ? have you use XMLBeanFactory ?**

**Ans**: In Spring Framework,

* A Bean Factory is a mechanism responsible for managing and providing beans in a Spring application.
* It's the central container where beans are defined and configured, and from which they are retrieved when needed by other components.
* The Bean Factory is responsible for the instantiation, configuration, and assembly of beans based on the metadata provided, such as XML configuration files or Java annotations.

**XMLBeanFactory:** This is the root interface for accessing spring bean container.

* XmlBeanFactory is a class provided by Spring Framework that implements the Bean Factory interface and specifically reads bean definitions from XML-based configuration files.
* It's part of the legacy Spring IoC container and is commonly used in older Spring applications where XML-based configuration is prevalent.

🡪 We don’t use this now.

While **XmlBeanFactory** is still supported in Spring, it's generally considered legacy, and the **ApplicationContext** interface, which provides more features and capabilities, is recommended for most modern Spring applications.

**Que: What is Application Context?**

**Ans:** In Spring Framework,

* The Application Context is the central interface for providing configuration information to an application. It represents the Spring IoC container and is responsible for managing the lifecycle of beans, providing dependency injection, and handling other application-level configuration tasks.
* It acts as a container that holds all the beans defined in the application and manages their creation, initialization, wiring, and destruction.
* The Application Context interface extends the Bean Factory interface and adds additional functionality and features, making it a more powerful and flexible container.

**Que: Differences between ApplicationContext and XmlBeanFactory?**

**Ans:**

**Bean Factory:**

* Bean instantiation and wiring.
* Lacks Cashing
* Limited to XML Files

**Application Context:**

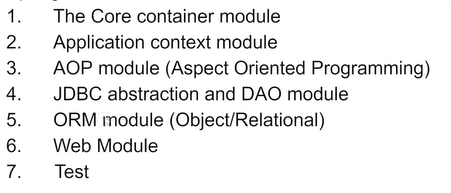
* Internationalization🡪(Abb as i18n)
* AOP Integration🡪
* Resource Loading🡪 accessing resource
* Automatic Bean Postprocessor Registration
* Caching Mechanism
* Load s XML, Java Class Annotations

Note :

In the context of software development, internationalization (often abbreviated as i18n, where 18 represents the number of letters between "i" and "n") refers to the process of designing and developing software applications in a way that allows them to be adapted easily to different languages, regions, and cultures.

**Que: What are the different module in Spring?**

**Ans**: Spring has seven core modules:



Spring Framework is a comprehensive framework for building enterprise Java applications. It is modular and consists of several modules, each addressing different aspects of application development. Here are the main modules of Spring Framework:

**1. Core Container:** This module provides the fundamental functionality of the Spring framework. It includes features such as dependency injection, inversion of control (IoC), bean lifecycle management, and configuration management. Key components include:

* **Spring Core**: Provides core utilities for the framework, such as the IoC container and resource management.
* **Spring Beans:** Provides the BeanFactory, which is the backbone of the Spring Framework for managing beans.
* **Spring Context**: Builds on the core and beans modules to provide application context functionality, including support for internationalization, event propagation, and resource loading.
* **Spring Expression Language (SpEL):** Provides a powerful expression language for querying and manipulating objects during runtime.

**2. Data Access/Integration:** This module provides support for data access and integration with databases, including JDBC, ORM frameworks like Hibernate, and data access templates. Key components include:

* **Spring JDBC**: Simplifies JDBC usage and helps manage database resources more efficiently.
* **Spring ORM:** Provides integration with ORM frameworks like Hibernate, JPA, and JDO for object-relational mapping.
* **Spring Transaction:** Supports declarative transaction management for JDBC, Hibernate, and JPA.

**3. Web:** This module provides support for building web applications using the Spring Framework. It includes features such as MVC architecture, RESTful web services, and WebSocket support. Key components include:

* **Spring Web MVC**: Implements the MVC design pattern for building web applications.
* **Spring Web:** Provides basic web-oriented integration features, such as multipart file upload, WebSocket support, and Servlet support.
* **Spring Webflux:** Provides support for reactive programming and building asynchronous, non-blocking web applications.

**4. AOP (Aspect-Oriented Programming):** This module provides support for aspect-oriented programming, allowing developers to define cross-cutting concerns separately from the core business logic. Key components include:

* **Spring AOP**: Implements AOP concepts such as aspect, advice, joinpoint, and pointcut within the Spring Framework.
* **Spring Aspects:** Provides integration with AspectJ, a powerful AOP framework.

**5. Messaging:** This module provides support for messaging systems and message-driven architectures. Key components include:

* **Spring Messaging:** Provides abstractions and support for messaging architectures, including support for STOMP over WebSocket, messaging annotations, and integration with message brokers like RabbitMQ and Apache Kafka.

**6. Testing:** This module provides support for testing Spring applications using unit tests, integration tests, and mock objects. Key components include:

**Spring Test:** Provides support for testing Spring applications using JUnit, TestNG, and other testing frameworks. It includes features such as dependency injection in tests, transaction management, and integration testing support.

**Que: Difference Between @Autowiring and @Inject?**

**Ans:** The @Inject Annotation also serves the same purpose as @Autowire.

The main difference between them is that @Inject is a Standard annotation for dependency injection and @Autowired is Spring Framework specific.

**Que: Difference between @Bean and @Component annotation is Spring?**

**Ans:**

**@Component** preferable for component scanning and automatic wiring.

**@Bean Annotation**: Returns an object that spring should register as bean in application context. The body of the method bears the logic responsible for creating the instance.

**@Bean:** The @Bean annotation is used to explicitly declare individual beans in a Spring configuration class. It allows you to define custom beans and configure their creation and initialization explicitly within Java configuration classes.

**@Component:** The @Component annotation is a generic stereotype annotation used to indicate that a class is a Spring component. It is typically used to auto-detect and automatically register beans during component scanning.

🡪 kabhi bhi khud ka logic likhna hai to @bean se likho.

**Que: What is Autowiring in Spring? What are the autowiring modes?(Different types of autowiring)**

**Ans:**

In Spring Framework, autowiring is a feature that allows the Spring container to automatically inject dependencies into Spring beans without explicitly specifying them in the bean configuration.

When a bean with dependencies is configured in the Spring container, autowiring examines the dependencies of the bean and tries to find matching beans from the container. If a matching bean is found, it is automatically injected into the dependent bean.

The autowiring modes in Spring are as follows:

**1. No Autowiring (default):**

* In this mode, autowiring is disabled, and dependencies must be explicitly configured using setter methods, constructor arguments, or field injection.
* This is the default mode, and no special annotations or configurations are required.

**2. By Type:**

* In this mode, Spring looks for a single bean of the same type as the dependency being injected. If exactly one matching bean is found, it is injected into the dependent bean.
* If multiple beans of the same type are found, Spring throws an exception, as it cannot determine which bean to inject.
* To use this mode, annotate the dependency with @Autowired or specify autowire="byType" in XML configuration.

**3. By Name:**

* In this mode, Spring looks for a bean with a matching name for the dependency being injected. If a bean with the same name is found, it is injected into the dependent bean.
* To use this mode, annotate the dependency with @Autowired and specify the bean name using @Qualifier annotation, or specify autowire="byName" in XML configuration.

**4. Constructor:**

* In this mode, Spring attempts to resolve dependencies by matching constructor arguments with beans in the container. If a constructor argument type matches a bean in the container, it is automatically injected.
* This mode is particularly useful for constructor injection and ensures that all dependencies required for object creation are satisfied.
* To use this mode, annotate the constructor with @Autowired, or specify autowire="constructor" in XML configuration.

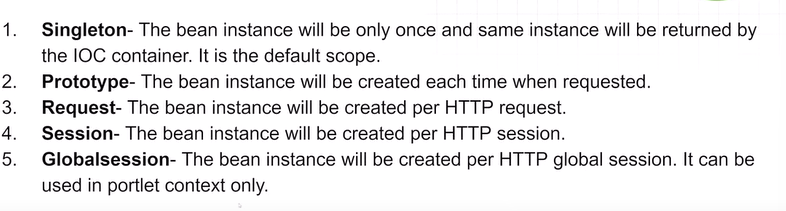
**Que: What are the different bean scopes in Spring?**

**Ans:** Singleton 🡪 eek bar Banega aur wo hi bar bar call hoga.

Prototype🡪 har bar eek naya bean Banega jab- jab request hoga.

Request🡪 HTTP request jitni bar ayegi wo create hoga.

Session🡪 Jitni bar naya http session Banega wo naya bean Banega. agr session wahi hai toh wahi purana wala call hoga bar bar.



**Que: What is Spring DevTools used for?**

**Ans**: When you are running spring boot applications. If you make changes to your source code. You have to manually restart your application.

Here is a rescue!

* Spring DevTools is tool that makes developing the applications faster and easier.
* It automatically restarts our application when we change code, so we can the updates immediately without restarting manually.
* It also refreshes our web browser automatically if we change in HTML files.
* Spring Boot framework to enhance the development experience for developers.
* Some of the key features of Spring Developer Tools include:

a. Automatic Application Restart

b. Live Code Reloading

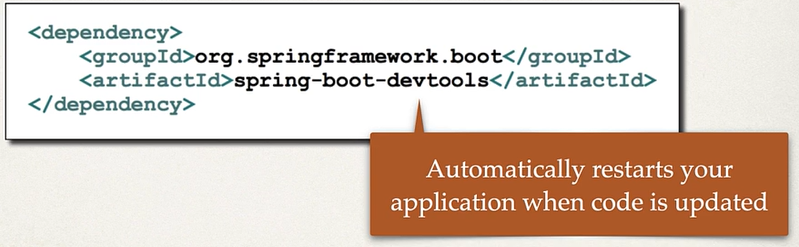
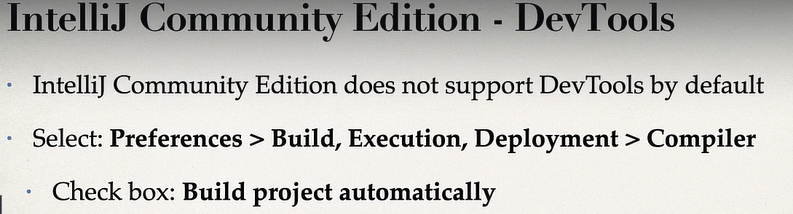
c. Remote Application Debugging: applications running remotely on a server or in a container.

d. Built-in Developer Console

e. Embedded LiveReload Server: Refreshes web pages if detect changes in html css.

**Que: How to enable DevTools in Your Project?**

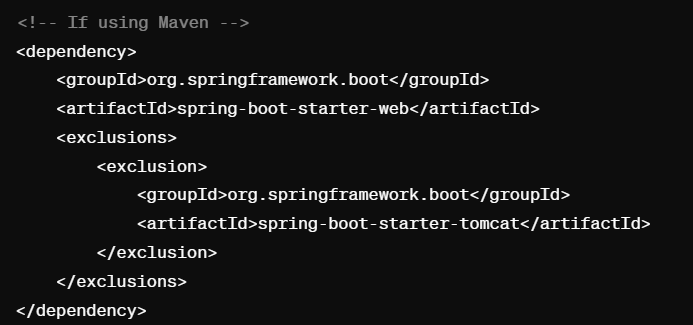
**Ans:** Simply add the dependency in pom. xml File.

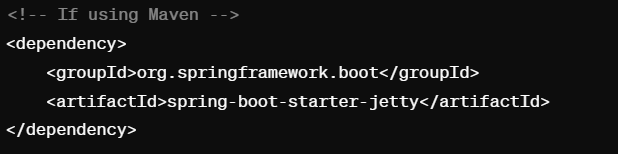
**Que: How to override or replace Tomcat Server in Spring Boot?**

**Ans**: To override or replace the embedded Tomcat server provided by default in a Spring Boot application, you can exclude Tomcat from the classpath and include an alternative servlet container or server implementation.

* Simply go to pom.xml or build.gradle, exclude the Tomcat dependency from Spring Boot Starter as its contained in that.



* Include Alternative Servlet Container: Add the dependency for the desired alternative servlet container.



* If necessary, then add configuration in application. properties.
* Spring boot will automatically configure the new server as the embedded server for you application.
* This flexibility allows us to choose the server that best suits our application.

**Que: Can we disable the default web server in the Spring Boot application?**

**Ans:** Yes, we can disable the default web server in the Spring Boot application. To do this, we need to set the server.port property to “-1” in the application’s application.properties file.

**Que: How to make spring boot know other components?**

**Ans:** To make Spring Boot aware of other components in your application, you typically need to ensure that those components are properly configured and managed by the Spring framework.

**Spring Boot Application Class:**

* Ensure that your main application class is annotated with @SpringBootApplication or a combination of @Configuration, @EnableAutoConfiguration, and @ComponentScan.
* This annotation enables Spring Boot's auto-configuration and component scanning features, allowing it to detect and manage your custom components.

**Component Scanning:**

* Spring Boot automatically scans for components (such as beans, controllers, services, repositories, etc.) within specific packages of your application. By default, it scans the package where your main application class is located and its sub-packages.

**Annotations:**

* Annotate your custom components with appropriate Spring annotations to indicate their roles and behaviors. For example, annotate service classes with @Service, repository classes with @Repository, and controller classes with @Controller or @RestController.

**Package Structure:**

* Organize your custom components in a structured package hierarchy. This not only helps with component scanning but also improves the maintainability and readability of your codebase.

**Dependency Injection:**

* Make use of dependency injection to wire your components together. Use @Autowired annotation or constructor injection to inject dependencies into your components.

**Que: What is @RequestMapping Annotations?**

**Ans:**

* In Spring MVC, request mapping is a mechanism used to map incoming HTTP requests to specific handler methods in your controllers.
* It can be applied at the class level to define a common base URI for all handler methods in the controller, as well as at the method level to define specific URI patterns for individual handler methods.

**Que: What is REST API?(what are RESTful Services?)**

**Ans:**

* REST API (Representational State Transfer Application Programming Interface) is an architectural style for designing networked applications.
* It defines a set of guidelines and principles for creating web services that are scalable, interoperable, and easy to maintain.
* RESTful APIs use standard HTTP methods (GET, POST, PUT, DELETE, PATCH, etc.) to perform operations on resources.
* Each HTTP method corresponds to a specific action on a resource:
* GET: Retrieve a representation of the resource.
* POST: Create a new resource.
* PUT: Update an existing resource or create a new resource if it doesn't exist.
* DELETE: Delete a resource.
* PATCH: Partially update an existing resource.
* OPTIONS, HEAD, etc.: Additional HTTP methods for metadata retrieval, checking resource existence, etc.

**Que: What are Rest End Points? (What are the end point you have used in your project)**

**Ans:** Endpoints are specific URLs (Uniform Resource Locators) or URIs (Uniform Resource Identifiers) that are exposed by a web service or web application to allow clients to interact with the service or application.

**@GetMapping**, **@PostMapping**, **@PutMapping**, **@DeleteMapping**, and **@PatchMapping** are specialized annotations.

**1. @GetMapping:**

* @GetMapping is used to map HTTP GET requests to handler methods in Spring MVC controllers.
* It is a shortcut for @RequestMapping(method = RequestMethod.GET)

**2. @PostMapping**:

* @PostMapping is used to map HTTP POST requests to handler methods in Spring MVC controllers.
* It is a shortcut for @RequestMapping(method = RequestMethod.POST).

**3. @PutMapping:**

* @PutMapping is used to map HTTP PUT requests to handler methods in Spring MVC controllers.
* It is a shortcut for @RequestMapping(method = RequestMethod.PUT).

**4. @DeleteMapping:**

* @DeleteMapping is used to map HTTP DELETE requests to handler methods in Spring MVC controllers.
* It is a shortcut for @RequestMapping(method = RequestMethod.DELETE).

**5. @PatchMapping:**

* @PatchMapping is used to map HTTP PATCH requests to handler methods in Spring MVC controllers.
* It is a shortcut for @RequestMapping(method = RequestMethod.PATCH).

**Que: What is the difference between RequestMapping and GetMapping?**

**Ans:** RequestMapping can be used with GET, POST, PUT, and many other request methods using the method attribute on the annotation. Whereas getMapping is only an extension of RequestMapping which helps you to improve on clarity on request.

**Que: Can we create a non-web application in Spring Boot?**

**Ans**: Yes, we can create a non-web application in Spring Boot. Spring Boot is not just for web applications. Using Spring Boot, we can create applications like Microservices, Console applications, and batch applications. we can create a non-web application by removing the web dependencies from the classpath along with changing the way Spring Boot creates the application context.

**Que: Is it possible to change the port of the embedded Tomcat server in Spring Boot?**

**Ans:** Yes, it is possible. By using the server.port in the application.properties.

**Que: What is the default port of tomcat in spring boot?**

**Ans:** The default port of the tomcat server-id 8080. It can be changed by adding sever.port properties in the application.property file.

**Que: Can we disable the default web server in the Spring boot application?**

**Ans**: Yes, we can use application. properties to configure the web application type i.e spring.main.web-application-type=none.

**Que: How to disable a specific auto-configuration class?**

**Ans:** You can use exclude attribute of @EnableAutoConfiguration if you want auto-configuration not to apply to any specific class.



**Que: Difference Between Controller and the REST Controller?**

**Ans:** In Spring Boot, @RestController and @Controller are both annotations used to define controller classes, but they have different purposes and behaviors.

**@Controller:**

* @Controller is a general-purpose annotation used to mark classes as Spring MVC controllers.
* Controllers annotated with @Controller are responsible for handling HTTP requests and generating appropriate responses.
* Typically, @Controller classes return a logical view name or a ModelAndView object, which resolves to a view template (HTML, JSP, Thymeleaf, etc.) that gets rendered on the client side.

**@RestController:**

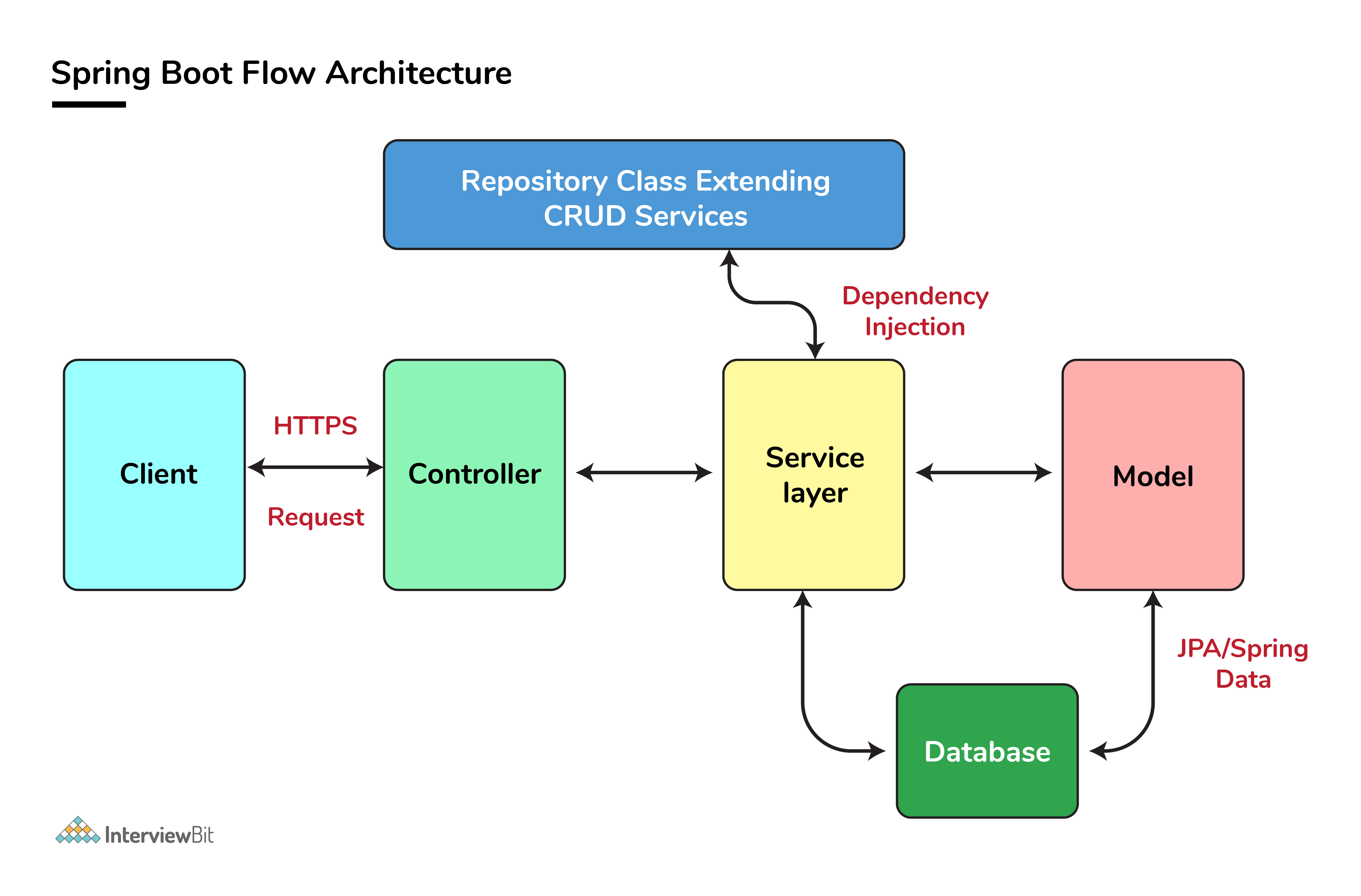
* @RestController is a specialized version of @Controller that is used to define RESTful web services.
* Controllers annotated with @RestController are responsible for handling HTTP requests and generating responses that are directly written to the HTTP response body.
* @RestController eliminates the need for annotating methods with @ResponseBody, as it combines @Controller and @ResponseBody annotations.
* By default, @RestController methods return the response in the JSON or XML format, depending on the Accept header of the request.

**Summary:**

* @Controller Map of the model object to view or template and make it human readable but @RestController simply returns the object and object data is directly written in HTTP response as JSON or XML.

**Que: Describe the flow of HTTPS requests through the Spring Boot application?**

**Ans:**



* First client makes an HTTP request (GET, POST, PUT, DELETE) to the browser.
* After that the request will go to the controller, where all the requests will be mapped and handled.
* After this in Service layer, all the business logic will be performed. It performs the business logic on the data that is mapped to JPA (Java Persistence API) using model classes.
* In repository layer, all the CRUD operations are being done for the REST APIs.
* A JSP page is returned to the end users if no errors are there.

**Que: What is the use of Profiles in spring boot?**

**Ans:** While developing the application we deal with multiple environments such as dev, QA, Prod, and each environment requires a different configuration. For e.g., we might be using an embedded H2 database for dev but for prod, we might have proprietary Oracle or DB2. Even if DBMS is the same across the environment, the URLs will be different.

* You define sets of configurations (like database URLs) for different situations (development, testing, production).
* Use the @Profile annotation to clarify which config belongs to where.
* Activate profiles with environment variables or command-line options.
* To use Spring Profiles, we simply need to define the spring.profiles.active property to specify which profile we want to use.

**Que: What are the different types of Environments?**

**Ans:** Multiple deployment environments refer to the various stages or environments in which a software application is deployed and run throughout its lifecycle. These environments typically include:

**1. \*\*Development Environment\*\*:**

* - Used by developers for writing and testing code locally.
* - May include local development servers, such as localhost or Docker containers, to simulate the production environment.

**2. \*\*Testing Environment\*\*:**

* - Used for testing new features, bug fixes, and changes in a controlled environment.
* - Includes various testing environments such as unit testing, integration testing, and system testing.

**3. \*\*Staging Environment\*\*:**

* - Mimics the production environment closely and is used for final testing before deploying changes to production.
* - Allows testing in an environment that closely resembles the production environment to identify potential issues before deployment.

**4. \*\*Production Environment\*\*:**

* - The live environment where the application is accessed and used by end-users.
* - Requires high availability, scalability, and reliability to handle real-world traffic and usage.

**5. \*\*Disaster Recovery (DR) Environment\*\*:**

* - A backup environment that is used in case of a disaster or failure in the production environment.
* - Provides redundancy and failover capabilities to ensure continuity of service in case of unexpected events.

**6. \*\*Quality Assurance (QA) Environment\*\*:**

* - Similar to the staging environment, used for testing and quality assurance purposes.
* - May include additional tools and processes for QA testing and validation.

**7. \*\*User Acceptance Testing (UAT) Environment\*\*:**

* - Used for user acceptance testing, where end-users validate new features or changes before deployment to production.
* - Provides a real-world testing environment for end-users to provide feedback on the application.

Each deployment environment serves a specific purpose in the software development lifecycle and helps ensure the quality, stability, and reliability of the application before it is deployed to production. The use of multiple environments helps minimize risks and ensure a smooth transition of changes from development to production.

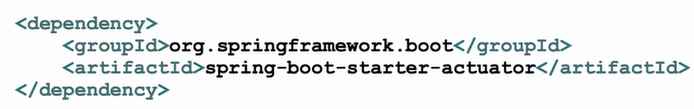
**Que: What is Spring Actuator? What are its advantages?**

**Ans**: An actuator is an additional feature of Spring that helps you to monitor and manage your application when you push it to production. These actuators include auditing, health, CPU usage, HTTP hits, and metric gathering, and many more that are automatically applied to your application.

***Actuator endpoints let you monitor and interact with your application. Spring Boot includes a number of built-in endpoints and lets you add your own. For example, the health endpoint provides basic application health information.***

**Que: How to enable Actuator in Spring boot application?**

**Ans:** To enable the spring actuator feature, we need to add the dependency of “spring-boot-starter-actuator” in pom.xml.

F

**Que: What are the actuator-provided endpoints used for monitoring the Spring boot application?**

**Ans:** Actuators provide below pre-defined endpoints to monitor our application -

* Health
* Info
* Beans
* Mappings
* Configprops
* Httptrace
* Heapdump
* Threaddump
* Shutdown

**Que: How to get the list of all the beans in your Spring boot application?**

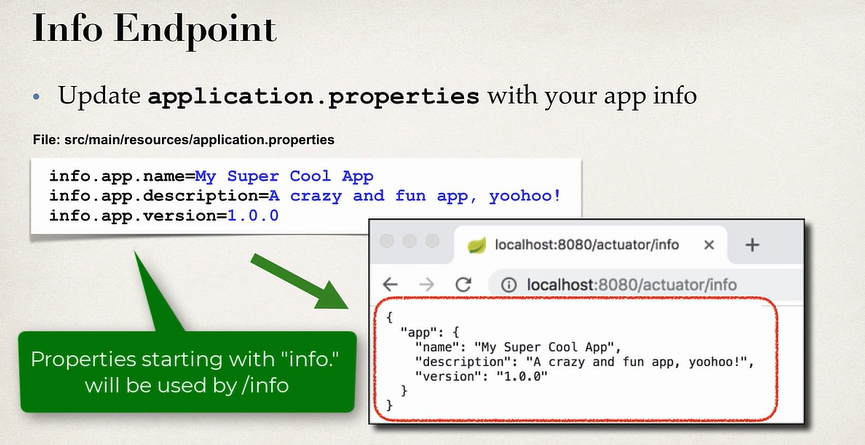
**Ans:** Spring Boot actuator “/Beans” is used to get the list of all the spring beans in your application.

**Que: How to check the environment properties in your Spring boot application?**

**Ans:** Spring Boot actuator “/env” returns the list of all the environment properties of running the spring boot application.

**Que: Where do we define properties in the Spring Boot application?**

**Ans:** You can define both application and Spring boot-related properties into a file called application. properties. You can create this file manually or use Spring Initializer to create this file. You don’t need to do any special configuration to instruct Spring Boot to load this file. If it exists in classpath then spring boot automatically loads it and configure itself and the application code accordingly.



**Que: What is Spring Initializr?**

**Ans:** Spring Initializer is a tool that helps us to create skeleton of spring boot project or project structure by providing a maven or gradle file to build the application. It set up the framework from scratch.

**Que: What are Spring Boot Properties (application.properties)?**

**Ans:**

**Spring Boot Properties**

* Spring boot can be configured in the ***application.properties*** file.
* Server port, context path, actuator, security, etc.
* Spring boot have 1000+ properties.



**Que: What is Lazy Initialization in Spring boot?**

Ans: In the context of Spring Boot and Java development, lazy initialization refers to the delayed creation or instantiation of Spring beans until they are actually needed by the application. Spring Boot provides support for lazy initialization of beans, which can be beneficial for improving application startup time, reducing resource consumption, and optimizing performance. Let's explore lazy initialization in Spring Boot:

**Enable Lazy Initialization:**

* In Spring Boot, lazy initialization can be enabled globally for all beans or selectively for specific beans using configuration properties or annotations.

**1. Global Lazy Initialization:**

You can globally enable lazy initialization for all Spring beans by adding the following property to your application.properties or application.yml file:

***spring.main.lazy-initialization=true***

**2. Selective Lazy Initialization:**

For selective lazy initialization of beans, you can use the @Lazy annotation on individual bean definitions:

***@Bean***

***@Lazy***

***public MyBean myBean() {***

***return new MyBean();***

***}***

***}***

**3. Lazy Initialization in Component Scanning:**

When using component scanning in Spring Boot (e.g., with @ComponentScan), you can control lazy initialization behavior for scanned beans:

***@ComponentScan(basePackages = "com.example", lazyInit = true)***

***public class MyConfiguration {***

***// Configuration code***

***}***

**Benefits of Lazy Initialization:**

* **Improved Startup Time:** Lazy initialization can significantly reduce the time taken for application startup, especially when dealing with a large number of beans or complex configurations.
* **Reduced Memory Consumption**: Beans that are not immediately needed are not instantiated upfront, leading to lower memory usage during application initialization.
* **Optimized Resource Usage**: By initializing beans on-demand, Spring Boot optimizes resource utilization and avoids unnecessary processing.

**Que: How to enable debugging log in the spring boot application?**

**Ans:** Debugging logs, often referred to as debug logs, are messages or information generated by software applications during runtime to provide detailed insights into the internal workings of the application. Debug logs are primarily used for troubleshooting, identifying issues, and understanding the flow of execution within the application.

To enable debugging log in Spring Boot Application, follow the below steps:

* Add the logging level property to application.properties.

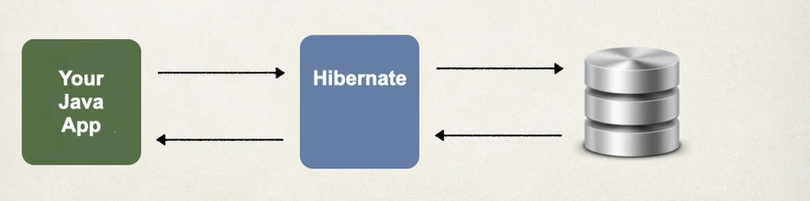
**logging.level.root=DEBUG**

* Configure the log pattern to include useful information.
* Run the Spring Boot application.
* Console Output: When you run your Spring Boot application, debugging logs will be displayed in the console alongside other log levels.
* Log Files: Spring Boot generates log files (e.g., spring.log, application.log) in the logs directory of your application's working directory. You can open these log files to view debugging logs along with other log levels.

**Hibernate and JPA Interview Question**

**Que: What are Hibernate?**

**Ans:** Hibernate is a Java-based persistence framework and an object-relational mapping (ORM) framework that basically allows a developer to map POJO - plain old Java objects - to relational database tables.



**Hibernate Maps:**

* **Java classes to database tables**.
* **Java objects to database records.**

**Hibernate:**

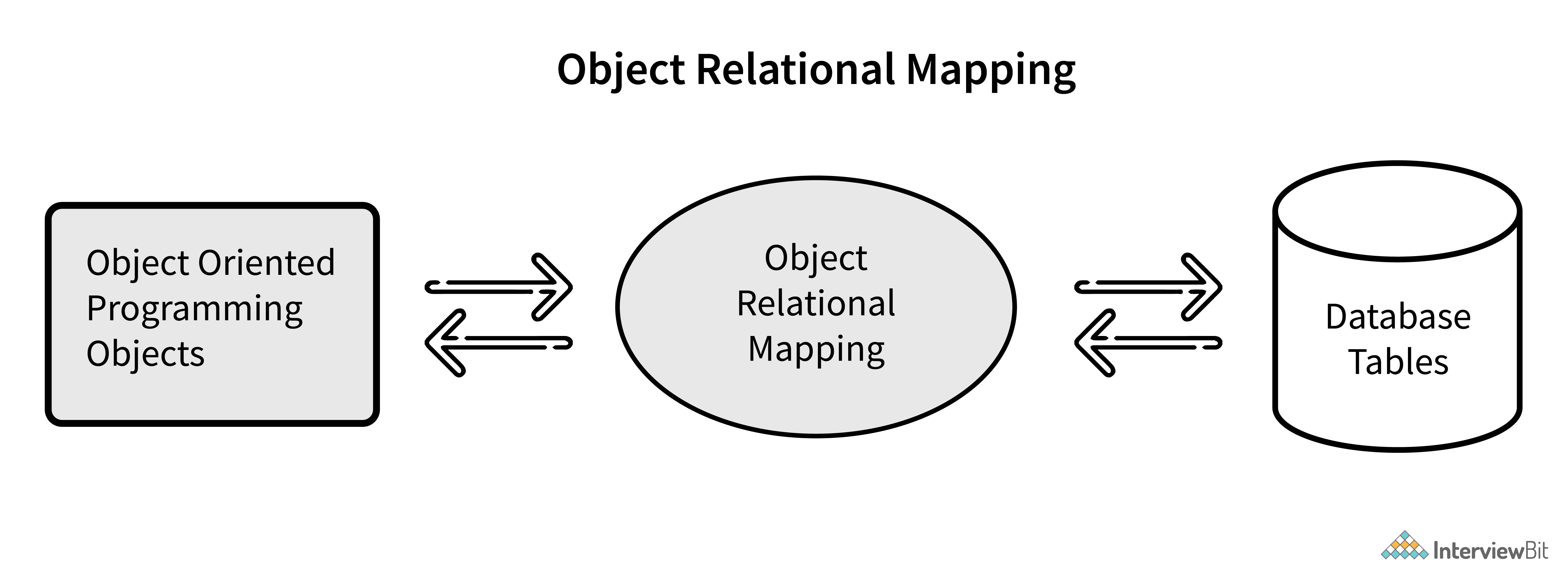
* Hibernate is a framework for persisting / saving Java Objects in a database.
* It used for saving and retrieving data from database.
* Hibernate is an open-source, object-relational mapping (ORM) framework for Java.
* It simplifies the interaction between Java applications and relational databases by **mapping Java classes to database tables** and **Java objects to database records.**
* Hibernate provides a powerful query language called Hibernate Query Language (HQL) for performing database operations.
* It offers features such as automatic schema generation, caching, and lazy loading to improve performance and developer productivity.
* Hibernate is widely used in Java enterprise applications for database access and persistence.

**Architecture:**



**Que: What is ORM in Hibernate?**

**Ans:** ORM (Object-Relational Mapping) in Hibernate refers to the technique of mapping Java objects to database tables and vice versa. Hibernate is an ORM framework for Java that provides a way to bridge the gap between object-oriented programming (OOP) and relational databases.



**1. Object-Relational Mapping (ORM):** It eliminates the need for developers to write SQL queries and manually manage the conversion between object-oriented data and relational data.

**2. Hibernate as an ORM Framework:** It provides a set of APIs and tools for mapping Java objects to database tables, performing CRUD (Create, Read, Update, Delete) operations, managing relationships between entities, and optimizing database access.

**3. Mapping Entities:** In Hibernate, entities are Java classes that represent objects mapped to database tables. Each entity class is annotated with metadata annotations (e.g., @Entity, @Table, @Column) to define its mapping to the corresponding database table and columns.

**4. SessionFactory and Session:** Hibernate uses a SessionFactory to create Session instances, which represent a single unit of work with the database. The Session interface provides methods for CRUD operations, querying data, managing transactions, and more.

**5. Object Identity and Persistence Context:** Hibernate manages the object lifecycle and object identity within a persistence context. When an entity is loaded or saved, it becomes associated with the current persistence context, allowing Hibernate to track changes and synchronize objects with the database.

**6. Querying with HQL and Criteria API**: Hibernate provides HQL (Hibernate Query Language) and the Criteria API for writing database queries using object-oriented syntax. HQL is similar to SQL but operates on entity objects, while the Criteria API allows for dynamic and type-safe querying.

**7. Caching and Performance Optimization:** Hibernate offers caching mechanisms, including first-level cache (session-level cache) and second-level cache (shared cache), to improve performance by reducing database roundtrips and optimizing data retrieval.

**Que: What are HQL?**

Ans: HQL stands for Hibernate Query Language. It is a powerful and flexible query language provided by Hibernate, which is an ORM (Object-Relational Mapping) framework for Java. HQL allows developers to write database queries using object-oriented syntax, making it easier to interact with databases and perform CRUD (Create, Read, Update, Delete) operations on entities. HQL is an object-oriented query language that operates on entity objects mapped to database tables. It abstracts away the details of SQL and allows developers to write queries using entity properties and relationships.

**Que: What are the advantages of Hibernate over JDBC?**

**Ans:** Hibernate offers several advantages over JDBC (Java Database Connectivity), especially in terms of developer productivity, code maintainability, and object-oriented database interactions. Here are some of the key advantages of Hibernate over JDBC:

1. **Object-Relational Mapping (ORM):**

* Hibernate provides a powerful ORM framework that allows developers to map Java objects to database tables and vice versa.
* This eliminates the need for writing complex SQL queries and manual data conversion code.

1. **Productivity and Rapid Development:**

* Hibernate significantly reduces development time by providing high-level abstractions and automatic mapping between Java objects and database tables.
* Developers can focus more on business logic and application functionality rather than dealing with low-level database interactions and SQL queries.

1. **Database Portability:**

* Hibernate abstracts the underlying database-specific SQL dialects and provides a uniform API for database access.
* This improves database portability and allows applications to work with different databases without major code changes.

1. **Automatic CRUD Operations:**

* Hibernate simplifies CRUD (Create, Read, Update, Delete) operations by providing built-in methods and APIs for saving, updating, querying, and deleting entities.
* Developers can perform database operations using Hibernate's session or transaction APIs without writing boilerplate JDBC code.

1. **Caching Mechanisms:**

* Hibernate includes caching mechanisms such as first-level cache (session-level cache) and second-level cache (shared cache) to improve performance and reduce database round-trips.
* Caching helps in storing and reusing frequently accessed data, resulting in faster response times and better scalability.

1. **Transaction Management:**

* Hibernate simplifies transaction management by providing transaction APIs that handle transaction boundaries, isolation levels, and automatic rollback in case of exceptions.
* Developers can manage transactions declaratively using annotations or programmatically using Hibernates transaction APIs.

1. **Integration with Java EE and Spring:**

* Hibernate integrates seamlessly with Java EE containers and Spring Framework, allowing for easy configuration and management of Hibernate sessions, transactions, and entities within enterprise applications.
* It supports various integration options such as JTA (Java Transaction API) for distributed transactions and Spring's declarative transaction management.

1. **Exception Handling:**

* Hibernate wraps the JDBC exceptions and throws unchecked exceptions like JDBCException or HibernateException.
* This along with the built-in transaction management system helps developers to avoid writing multiple try-catch blocks to handle exceptions.
* In the case of JDBC, it throws a checked exception called SQLException thereby mandating the developer to write try-catch blocks to handle this exception at compile time.

**Que: How does Hibernate Queries Database?**

**Ans:** Hibernate queries the database using a combination of techniques and mechanisms that leverage its ORM (Object-Relational Mapping) capabilities. When you execute a Hibernate query, such as an HQL (Hibernate Query Language) query or a Criteria API query, hibernate translates that query into corresponding SQL statements to interact with the database.

Certainly! Here's a summary of how Hibernate queries the database in points:

1. Hibernate queries are created using HQL (Hibernate Query Language), Criteria API, or native SQL queries.
2. HQL queries or Criteria API queries are translated into SQL statements by Hibernate based on the underlying database dialect.
3. Hibernate uses mapping metadata to generate SQL statements that fetch, filter, join, and aggregate data from the database.
4. Parameter binding allows dynamic values to be passed to queries safely.
5. The SQL statements are executed against the database using JDBC.
6. Result sets from the database are mapped back to Java objects (entities or custom result objects) based on the query's projection and mappings.
7. Hibernate includes caching mechanisms and query optimizations to improve performance and reduce database roundtrips.
8. The overall process abstracts away SQL complexities, providing a seamless and efficient way to interact with databases in Java applications.

**Que: What is a Session in Hibernate?**

**Ans:** In Hibernate, a Session represents a single unit of work with the database. A session is an object that manages the connection between Java objects (entities) and the database.

Session also has methods for storing, retrieving, modifying or deleting data from database using methods like persist (), load(), get(), update(), delete(), etc. Additionally, It has factory methods to return Query, Criteria, and Transaction objects.

***bean create hoga fir session create hoga for uss session me bean database ke sath map hoga aur jo operation karna hoga krega aur fir session end ho jayega.***

Here are key points about Sessions in Hibernate:

**Lifecycle of a Session:**

* A Session is typically created using a SessionFactory, which is responsible for managing database connections and providing Session instances.
* The lifecycle of a Session begins with its creation, continues with database operations (such as saving, updating, deleting entities), and ends with either committing or rolling back transactions.

**Session Management:**

* Hibernate manages the lifecycle of Sessions automatically within a managed environment (such as Java EE containers or Spring Framework).
* Sessions should be properly opened, used for database operations, and closed to release database resources and avoid resource leaks.

**Thread Safety:**

* Sessions in Hibernate are not thread-safe by default. Each Session should be used within the context of a single thread and not shared among multiple threads concurrently.
* However, Session Factory instances can be thread-safe and shared across multiple threads in a multi-threaded environment.

**Que: What is a SessionFactory?**

**Ans:** A SessionFactory in Hibernate is a crucial component responsible for creating and managing Session objects. It plays a central role in Hibernate's ORM (Object-Relational Mapping) framework by providing a factory pattern to produce Session instances.

***Jaise application on hota hai waise hi session factory b on ho jata hai .. iska life cycle tab tak hota hai jab tak application chalta hai ..jaise hi application band ye b band.. issi me session create hota hai kam karta hai aur end ho jata hai… yahi manage karta hai mapping and database connection.. ye thread safety ensure krta hai..***

Here are key points about SessionFactory in Hibernate:

**1. Session Creation:**

The SessionFactory is used to create Session instances, which represent a single unit of work with the database.

Sessions are typically short-lived and are created when needed for database operations.

**2. Database Connection Management:**

* The SessionFactory manages database connections and connection pooling behind the scenes.
* It establishes and maintains connections to the database, optimizing resource usage and improving performance by reusing connections when possible.

**3. Thread-Safe Initialization:**

* SessionFactory instances are thread-safe and should be created only once during application startup.
* It is common to configure the SessionFactory as a singleton bean in Java EE containers or Spring Framework to ensure thread safety and efficient resource utilization.

**4. Configuration and Metadata:**

* The SessionFactory is configured with Hibernate properties, such as database connection details, dialect, caching options, and mapping metadata.
* Mapping metadata includes information about entity classes, associations, primary keys, and database table mappings defined in Hibernate mappings (e.g., XML mappings or annotations).

**5. Session Cache Management:**

* The SessionFactory manages the second-level cache, which is a shared cache across multiple Session instances.
* The second-level cache stores persistent entities and their associated data, improving performance by reducing database round-trips and query execution overhead.

**6. Transaction Management:**

* SessionFactory instances provide transaction management capabilities for Hibernate sessions.
* They coordinate database transactions, transaction isolation levels, and transaction boundaries within the context of a Session's unit of work.

**7. Session Lifecycle:**

* A SessionFactory's lifecycle typically aligns with the application's lifecycle. It is created during application initialization and destroyed during application shutdown.
* Sessions are created and managed by the SessionFactory as needed and are closed after completing database operations or transaction boundaries.

**Que: What are Transaction Management in Hibernate?**

**Ans:** In Hibernate, transaction management is a critical aspect of ensuring data integrity, consistency, and reliability when interacting with databases. Transactions represent a series of database operations that should be treated as a single unit of work. Hibernate provides mechanisms for managing transactions effectively. Here's how transaction management works in Hibernate:

**1. Transaction Management Interfaces:**

* Hibernate provides interfaces for managing transactions, primarily through the org.hibernate.Session interface and the org.hibernate.Transaction interface.
* The Session interface represents a single unit of work with the database and provides methods for managing transactions, executing queries, and interacting with entities.

**2. Beginning a Transaction:**

* To start a transaction in Hibernate, you first obtain a Session object either directly from a SessionFactory or through dependency injection (e.g., in a Spring application).
* Once you have the Session object, you begin a transaction using the beginTransaction() method of the Session.

**3. Performing Database Operations:**

* Within the transaction scope, you can perform various database operations such as saving entities, updating data, deleting records, or executing queries.
* These operations are carried out using methods provided by the Session interface, such as save(), update(), delete(), createQuery(), etc.

**4. Committing or Rolling Back Transactions**:

* After executing the desired database operations within the transaction, you have the option to either commit the transaction or roll it back.
* Committing a transaction makes the changes permanent in the database, while rolling back undoes all changes made during the transaction.

**5. Exception Handling:**

* It's important to handle exceptions that may occur during transaction execution, such as database errors, constraint violations, or application-specific errors.
* Hibernate provides built-in exception types like HibernateException and JDBCException that you can catch and handle appropriately.

**6. Transaction Isolation Levels:**

* Hibernate allows you to configure transaction isolation levels to control the visibility of database changes among concurrent transactions.
* Common isolation levels include READ\_COMMITTED, REPEATABLE\_READ, SERIALIZABLE, etc., which determine the level of data consistency and concurrency control.

**Que: How does spring call a hibernate?**

**Ans**: Spring can call Hibernate through integration mechanisms provided by both frameworks. Here's an overview of how Spring interacts with Hibernate:

**1. Configuration:**

* First, you need to configure Hibernate in your Spring application. This includes setting up Hibernate configuration properties, defining entity classes, mapping metadata (either through XML mappings or annotations), and configuring a Hibernate SessionFactory.

**2. SessionFactory Bean:**

* In a Spring application, you typically create a Hibernate SessionFactory bean using Spring's configuration mechanisms. This bean represents the factory for creating Hibernate Session instances.
* The SessionFactory bean can be configured in XML configuration files or using Java-based configuration with annotations.

**3.Session Management:**

* Once the SessionFactory bean is created, Spring manages the lifecycle of Hibernate Session instances within the application context.
* You can obtain a Session instance either directly from the SessionFactory bean or using Spring's HibernateTemplate or HibernateDaoSupport classes.

**4. Transaction Management:**

* Spring provides transaction management capabilities that integrate with Hibernate's transaction support.
* You can use Spring's @Transactional annotation to mark methods or classes as transactional, and Spring manages transaction boundaries, commits, rollbacks, and exception handling.

**5. Dependency Injection:**

* Spring promotes dependency injection, allowing you to inject Hibernate components such as SessionFactory or Session into Spring-managed beans using annotations like @Autowired.

**6. Integration with Spring Data:**

* Spring Data JPA, a part of the Spring Data project, provides additional abstraction layers on top of Hibernate for simplified data access and repository management.
* Spring Data JPA integrates with Hibernate's EntityManager and JPA repositories, allowing you to use repository interfaces with automatic query generation and CRUD operations.

**Que: What are Caching?**

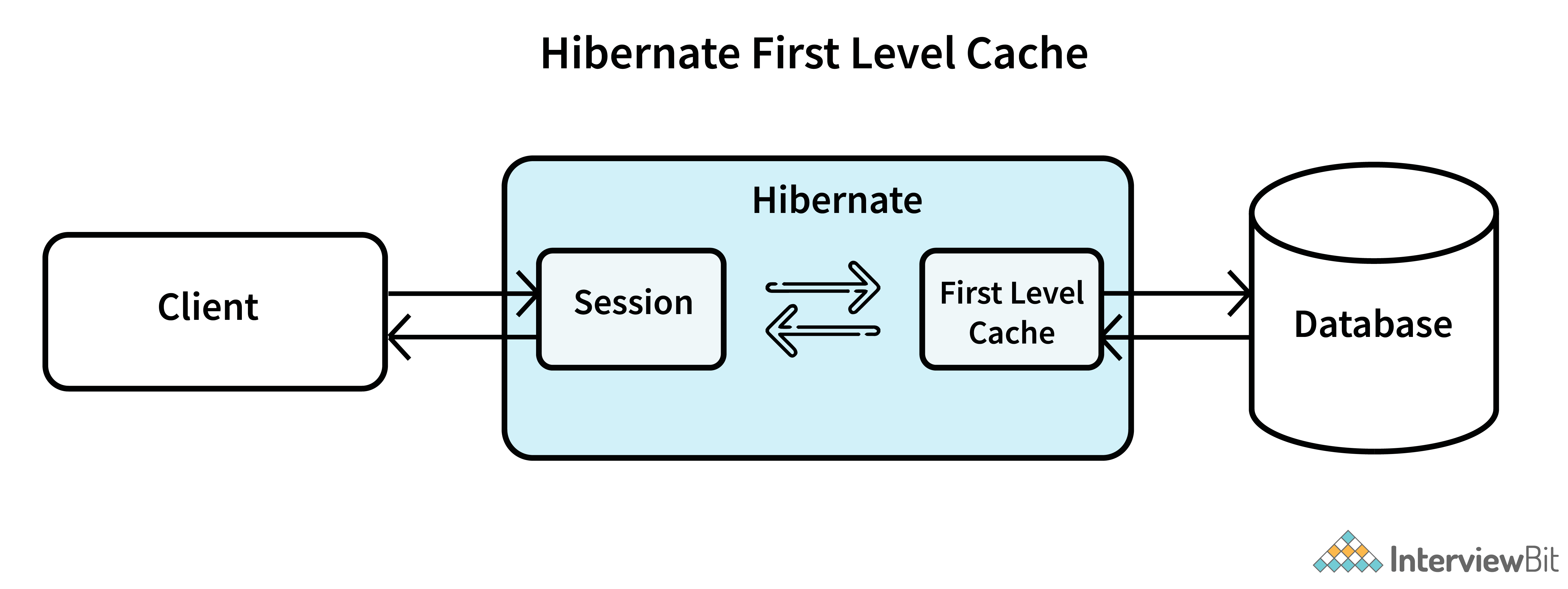
**Ans:** Caching is a mechanism used to improve performance by storing frequently accessed data in memory, reducing the need to fetch data repeatedly from the database.

Caching is like having a memory box where you can store things you use frequently, so we don't have to go through the whole process of getting them each time. It makes our application faster and more efficient.

Hibernate provides two levels of caching:

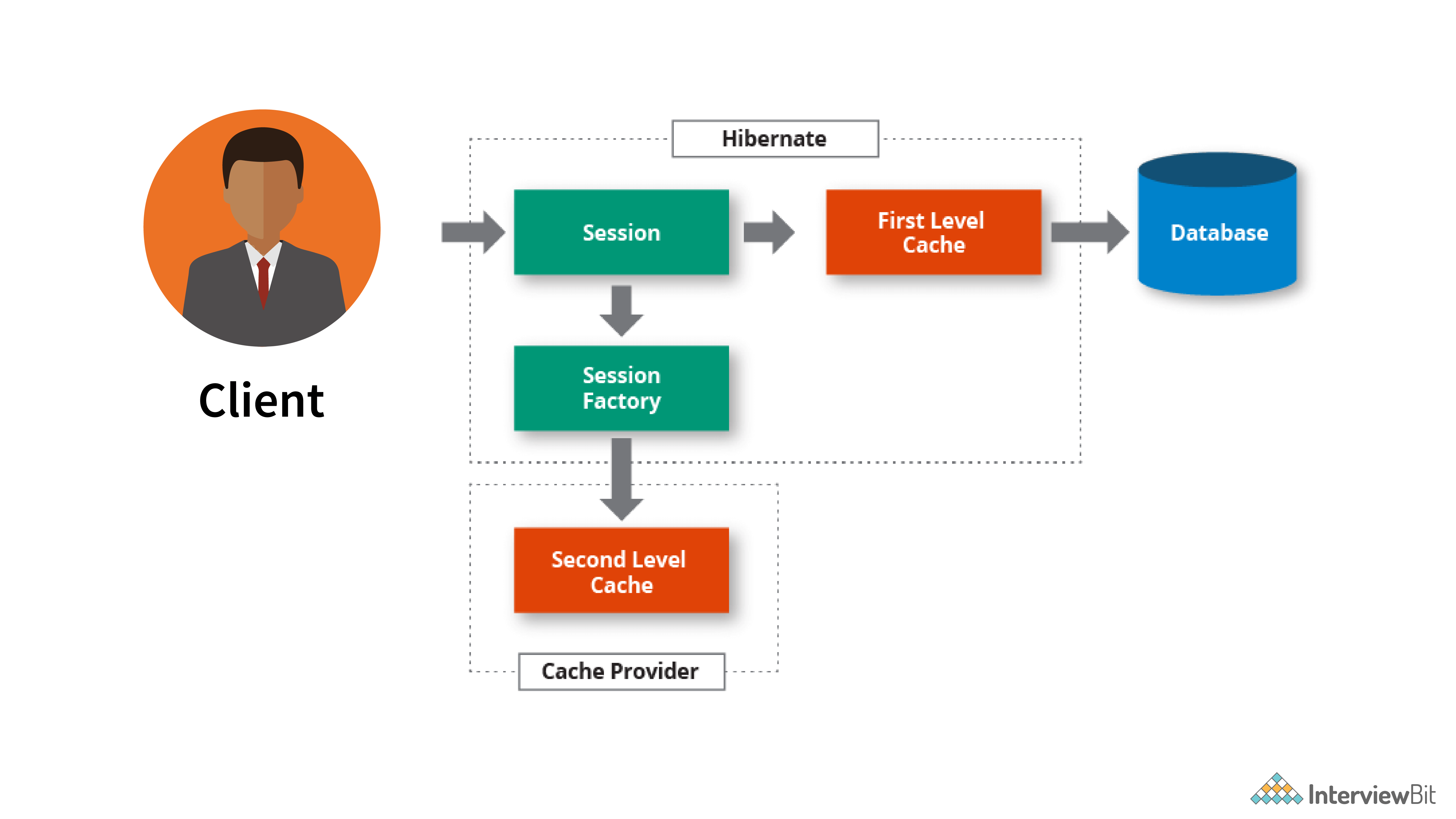
**Level 1 Caching:**

* **Scope**: Level 1 caching, also known as the first-level cache, operates at the session level.
* **Cache Provider:** It is provided by Hibernate's Session object.
* **Cache** **Content**: Level 1 cache stores entities and their associated data (persistent objects) that are currently associated with a particular Hibernate Session.
* **Cache** **Usage**: Whenever an entity is loaded or saved using a Session, hibernate checks the Level 1 cache first to see if the entity is already in memory. If found, it returns the cached object, reducing the need for repeated database queries for the same entity within the same session.
* **Lifecycle**: Level 1 cache is associated with the Hibernate Session and is cleared automatically when the session is closed or cleared explicitly.

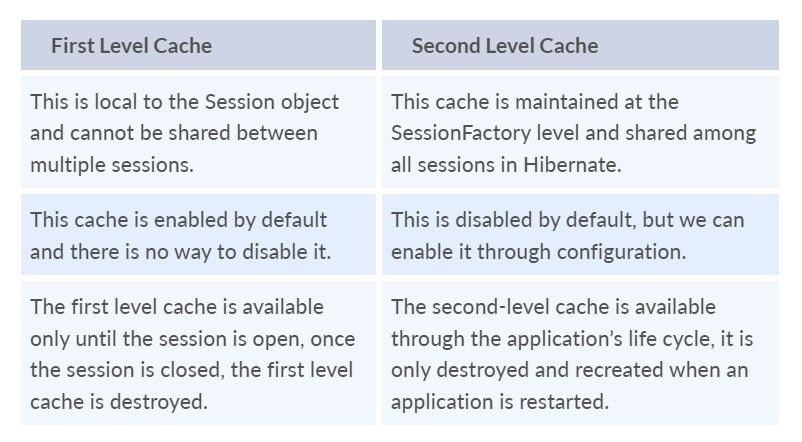


**Level 2 Caching:**

* **Scope**: Level 2 caching operates at a wider scope compared to Level 1 cache. It is also known as the second-level cache.
* **Cache** **Provider**: Level 2 cache is typically provided by external caching providers such as Ehcache, Redis, Hazelcast, or even distributed caching solutions.
* **Cache** **Content**: Level 2 cache stores entities and their associated data across multiple sessions within the application. It is shared across different sessions and transactions.
* **Cache** **Usage**: Level 2 cache helps in reducing database round-trips and improving performance by caching frequently accessed entities at a global level. When an entity is requested, Hibernate checks the Level 2 cache first before going to the database, if enabled and configured.
* **Lifecycle**: Level 2 cache is not tied to a specific session and can outlive individual sessions. It needs to be managed and configured separately from Hibernate's session management.



**Key Differences:**

****

**Que: What is Spring Cache Abstraction?**

**Ans:** The Spring Cache abstraction is a feature provided by the Spring Framework that simplifies the integration of caching mechanisms into Spring applications. It offers a declarative way to add caching functionality to methods, reducing the complexity of manual caching management.

**Key Concepts and Annotations**

* @**EnableCaching**: This annotation is used to enable caching support in a Spring Boot application. It should be placed on a configuration class or the main application class.
* @**Cacheable**: Applied to methods, this annotation specifies that the method's return value should be cached based on the specified cache name. Subsequent invocations of the method with the same arguments will return the cached value without executing the method logic again.
* @**CachePut**: Similar to @Cacheable, @CachePut stores the method's return value in the cache, but it always executes the method and updates the cache with the latest value.
* @**CacheEvict**: This annotation removes entries from the cache. It can be used to evict (expel- nikal dena) a single entry or clear the entire cache.
* @**CacheConfig**: This annotation allows you to specify default cache settings, such as the cache names to use, for a class. Methods within the class can then use these default settings unless overridden.

**Configuration in Spring Boot**

1. To use the Spring Cache abstraction in a Spring Boot application:
2. Add the spring-boot-starter-cache dependency to your project's dependencies. Spring Boot provides auto-configuration for common caching providers like Ehcache, Redis, Caffeine, etc.
3. Enable caching in your application by annotating your main application class (or a configuration class) with @EnableCaching.
4. Define cache-related properties in your application configuration (e.g., application.properties or application.yml). For example, you can specify the caching provider, cache names, TTL (time to live) settings, etc.
5. Use caching annotations (@Cacheable, @CachePut, @CacheEvict) in your service or repository methods to enable caching for specific operations.

**Que: How would you implement caching in a Spring Boot application?**

**Ans:** To implement caching in a Spring Boot application:

* first add a caching dependency, like spring-boot-starter-cache.
* Then, enable caching in the application by adding @EnableCaching annotation to the main class.
* Define cacheable operations using @Cacheable on methods whose results we want to cache. Optionally, customize cache behavior with annotations like @CacheEvict and @CachePut.
* Choose a cache provider (like EhCache or Hazelcast) or use the default concurrent map-based cache provided by Spring.

**Que: What is Lazy Loading?**

Ans: Lazy loading is a technique used in software development, particularly in the context of databases and object-relational mapping (ORM), to defer the loading of certain data until it is actually needed.

In the context of **Hibernate** (which is often used in Spring Boot applications), lazy loading refers to the delayed fetching of related entities or collections until they are accessed or referenced.

This signifies that child objects are not loaded until the parent gets loaded.

**For example**, consider a parent entity (e.g., Order) that has a lazy-loaded collection of child entities (e.g., OrderItem). When the Order entity is loaded, the OrderItem collection is not fetched until the application accesses it.

**Efficient Use of Resources:**

* Lazy loading helps in optimizing resource usage, especially when dealing with large datasets or complex object graphs.
* By loading data only when needed, lazy loading reduces memory consumption and minimizes the amount of data transferred between the application and the database.

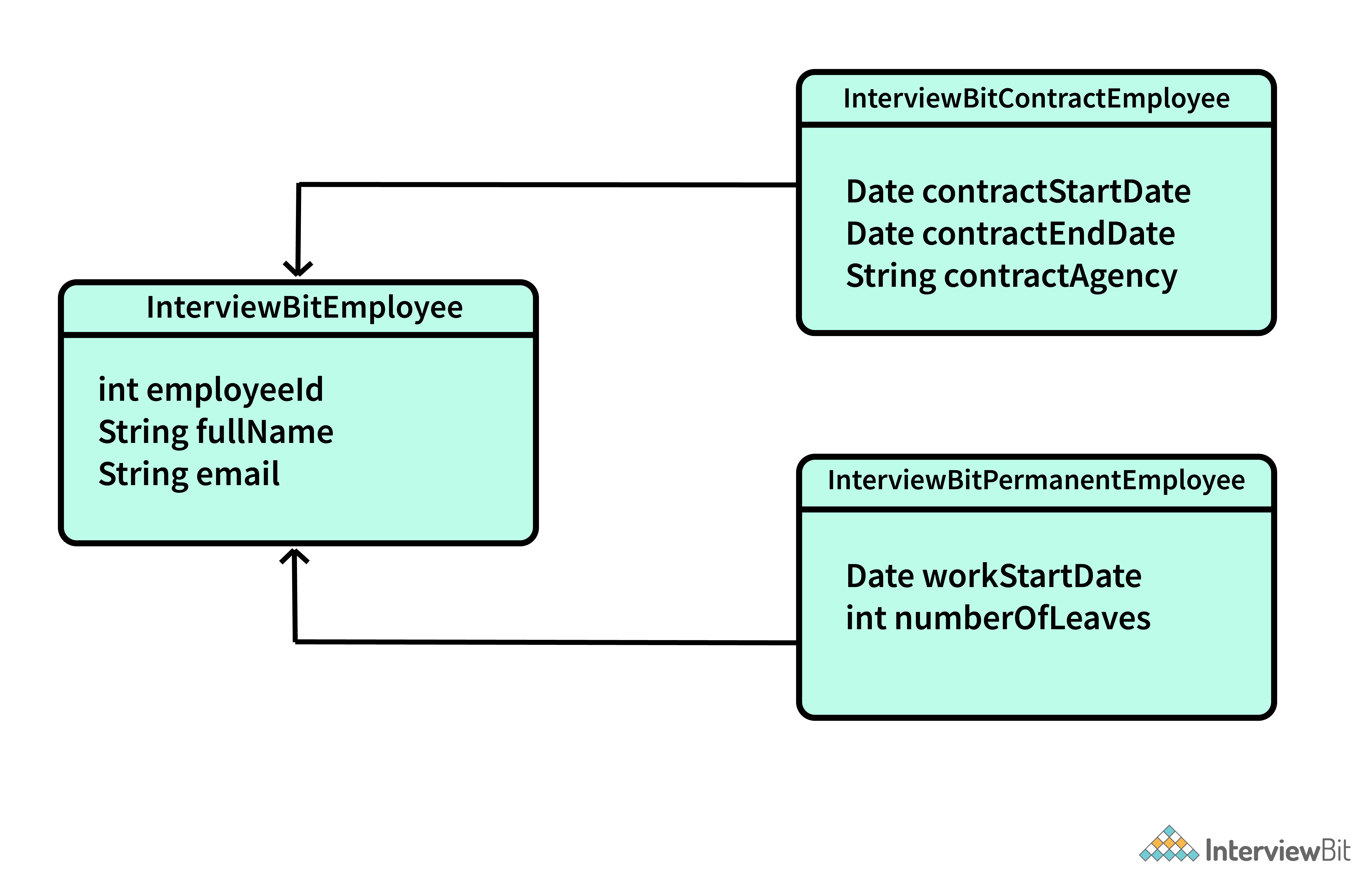
**Configuration:**

* Lazy loading behavior can be configured in Hibernate using annotations (@ManyToOne, @OneToMany, etc.) or XML mapping files.
* By default, many-to-one and one-to-one associations are lazy loaded, while one-to-many and many-to-many associations are eagerly loaded. You can override these defaults as per your application's requirements.

**Que: What is Hibernate Inheritance mapping?**

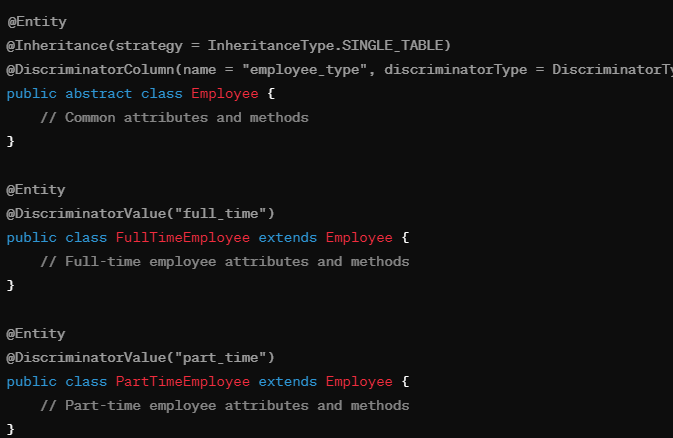
**Ans:** Hibernate inheritance mapping is a technique used to map inheritance relationships between Java classes onto corresponding database tables. It allows you to represent the hierarchical structure of your domain model in the database, taking advantage of object-oriented programming concepts like inheritance, polymorphism, and encapsulation.

There are several inheritance mapping strategies supported by Hibernate, each suited for different use cases and scenarios. The common inheritance mapping strategies in Hibernate are:



**1. Single Table Inheritance (STI):**

* In Single Table Inheritance, all classes in the inheritance hierarchy are mapped to a single database table.
* Hibernate adds a discriminator column to the table to differentiate between different types of entities.
* The discriminator column typically contains a value that identifies the subclass type (e.g., a string or an enumeration).



**2. Joined Table Inheritance:**

* In Joined Table Inheritance, each class in the inheritance hierarchy is mapped to a separate database table.
* Hibernate uses foreign key relationships between tables to represent the inheritance relationships.
* Each subclass table contains only its specific attributes, and a join query is used to retrieve data from multiple tables for a complete entity.

**3. Table Per Class Inheritance:**

* In Table Per Class Inheritance, each class in the hierarchy is mapped to its own database table.
* Hibernate duplicates common attributes from the superclass into each subclass table, resulting in denormalized data.
* This strategy can lead to redundant data storage but allows for efficient querying without joins.

**Que: How do you create an immutable class in hibernate?**

**Ans:** In Hibernate, creating an immutable class refers to defining a class whose instances cannot be modified once they are created.

**1. Define the Class:**

Start by defining your immutable class in Java. An immutable class should have the following characteristics:

* All fields should be declared as final to prevent modification.
* No setter methods should be provided for the fields.
* Fields should be initialized either through the constructor or initialization blocks.

**2. Hibernate Annotations:**

* Hibernate provides annotations to enforce immutability at the database level.
* The @Immutable annotation is used to specify that an entity is immutable and Hibernate should not attempt to modify it in the database.

**Que: What can you tell about Hibernate Configuration File?**

**Ans:** The Hibernate Configuration File (hibernate.cfg.xml or hibernate.properties) is a crucial component in a Hibernate-based application. It contains essential settings and configurations required by Hibernate to establish a connection with the database, manage entity mappings, handle transactions, and configure various Hibernate-specific properties.

* Hibernate supports configuration through both XML files (hibernate.cfg.xml) and properties files (hibernate.properties).
* XML configuration provides a more structured and detailed approach, allowing you to define various elements such as database connection details, entity mappings, cache settings, and more.
* Properties files offer a simpler and more concise way to configure Hibernate using key-value pairs, suitable for basic configurations.

**Que: Is hibernate prone to SQL injection attack?**

**Ans:** When using Hibernate correctly, the risk of SQL injection is significantly reduced compared to raw SQL queries. However, it's essential to follow best practices to ensure security.

Hibernate does not provide immunity to SQL Injection. However, following good practices avoids SQL injection attacks. It is always advisable to follow any of the below options:

The key points about Hibernate and SQL injection in short:

* Hibernate helps prevent SQL injection by using parameterized queries and automatic escaping of special characters.
* Parameterized queries separate SQL code from user input, reducing the risk of injection attacks.
* Hibernate encourages the use of named parameters and validates queries during parsing and compilation.
* Input validation and sanitization on the server side are crucial to further protect against SQL injection.
* Minimize dynamic SQL constructs and avoid concatenating user input directly into queries.
* Follow the least privilege principle to limit database permissions for enhanced security.

**Que: What are the most commonly used annotations available to support hibernate mapping?**

**Ans:** Here are some of the most commonly used annotations in Hibernate mapping:

**1. @Entity:**

Marks a Java class as an entity, indicating that instances of this class will be mapped to database tables.

**2. @Table:**

* Specifies the name of the database table to which an entity is mapped.
* Allows customization of table-related properties such as schema, catalog, and indexes.

***@Entity***

***@Table(name = "products")***

***public class Product {***

***// Class definition***

***}***

**3. @Id:**

* Marks a field or property as the primary key of the entity.
* The primary key can be auto-generated using different strategies such as identity, sequence, table, etc.

**4. @GeneratedValue:**

* Specifies the strategy used to generate values for the primary key.
* Strategies include GenerationType.IDENTITY, GenerationType.AUTO, GenerationType.SEQUENCE, GenerationType.TABLE, etc.

***@Id***

***@GeneratedValue(strategy = GenerationType.IDENTITY)***

***private Long id;***

**5. @Column:**

* Maps an entity's field or property to a specific database column.
* Allows customization of column-related properties such as name, length, nullable, unique, precision, scale, etc.

***@Column(name = "product\_name", nullable = false, length = 100)***

***private String productName;***

**6. @OneToOne, @OneToMany, @ManyToOne, @ManyToMany:**

* Define relationships between entities such as one-to-one, one-to-many, many-to-one, and many-to-many.
* Specify the target entity, join column(s), fetch type, cascade behavior, and other relationship attributes.

**7. @JoinColumn:**

* Specifies the join column(s) used in a relationship mapping.
* Allows customization of join column properties such as name, nullable, referencedColumnName, foreignKey, etc.

***@OneToOne***

***@JoinColumn(name = "category\_id", referencedColumnName = "id", nullable = false)***

***private Category category;***

**8. @Transient:**

* Marks a field or property as transient, indicating that it should be excluded from persistence (not mapped to a database column).
* Useful for properties that are derived or calculated and not stored in the database.

**@Transient**

**private transient String transientProperty;**

**jo calculate hota hai lekin database me store na ho jaise a= b+C**

**Que: Difference between getCurrentSession and openSession methods?**

**Ans:** In Hibernate, both getCurrentSession() and openSession() methods are used to obtain a Session object, which is essential for interacting with the database.

**getCurrentSession():**

**Usage:**

* getCurrentSession() is typically used in applications that manage sessions within a controlled environment, such as Java EE containers that support JTA (Java Transaction API) transactions.
* It is designed for applications that use the "session-per-request" or "session-per-transaction" pattern, where a new session is created and managed for each HTTP request or transaction.

**Session Management:**

* The getCurrentSession() method returns the current Hibernate session bound to the current thread context.
* It relies on external session context management, such as a transaction manager in Java EE environments or a custom session context provider in standalone applications.

**openSession():**

**Usage:**

* openSession() is more flexible and suitable for applications that require manual session management or custom transaction handling.
* It is commonly used in standalone applications or scenarios where session lifecycle control is required.

**Session Management:**

* The openSession() method creates a new Hibernate session instance every time it is called, providing a fresh session that is not bound to any specific context.
* It is the responsibility of the application to manage the lifecycle of the session, including opening, flushing, and closing the session explicitly.

**Que: Differentiate between save() and saveOrUpdate() methods in hibernate session.**

**Ans:**

* **save():** Used to insert a new record into the database. If the entity already has an identifier (primary key) set, save() will throw an exception. It returns the generated identifier after successful insertion.
* **saveOrUpdate():** Used to either save a new record or update an existing record in the database. If the entity has an identifier, saveOrUpdate() will perform an update; otherwise, it will perform an insert. It returns the identifier after saving or updating the entity.

**Que: Differentiate between get() and load() in Hibernate session**

**Ans:**

* get() returns the entity object immediately and may return null if the entity is not found.
* load() returns a proxy object and defers fetching until accessed, throwing an exception if the entity is not found when accessed. Use get() for immediate fetching and handling possible nulls, and load() for lazy loading and when sure of entity existence.

**Que: What does session.lock() method in hibernate do?**

**Ans:** The session.lock() method in Hibernate is used to reattach a detached entity to the current Hibernate session and acquire a lock on that entity. This method is particularly useful in scenarios where you have a detached entity (i.e., an entity that was previously associated with a session but is now not associated with any session) and you want to reattach it to the session for further operations.

**Que: When is merge() method of the hibernate session useful?**

**Ans:** Merge() method can be used for updating existing values. The specialty of this method is, once the existing values are updated, the method creates a copy from the entity object and returns it. This result object goes into the persistent context and is then tracked for any changes. The object that was initially used is not tracked.

**Que: What are types of Association in hibernate?**

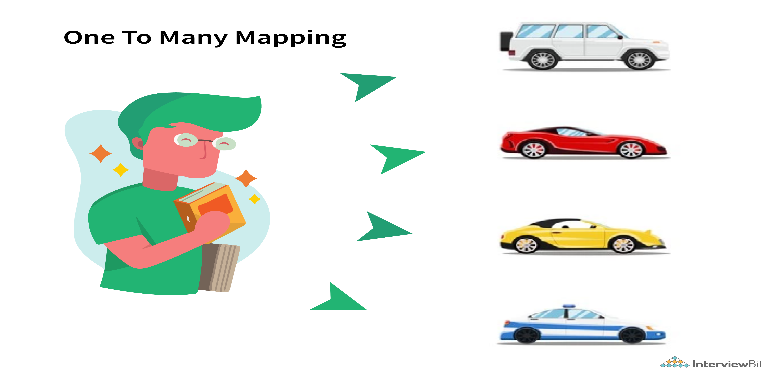
**Ans:** In Hibernate, associations define the relationships between entities (POJOs) in an object-oriented manner. There are several types of associations that you can define between entities based on how the entities are related to each other. Here are the main types of associations in Hibernate:

**1. One-to-One (1:1) Association**:

* In a one-to-one association, each record in one entity is associated with exactly one record in another entity, and vice versa.
* Example: A Person entity is associated with exactly one Passport entity, and each passport is associated with exactly one person.

**2. One-to-Many (1:N) Association:**

* In a one-to-many association, each record in one entity can be associated with multiple records in another entity, but each record in the second entity is associated with exactly one record in the first entity.
* Example: A Department entity can have multiple Employee entities associated with it, but each employee belongs to only one department.

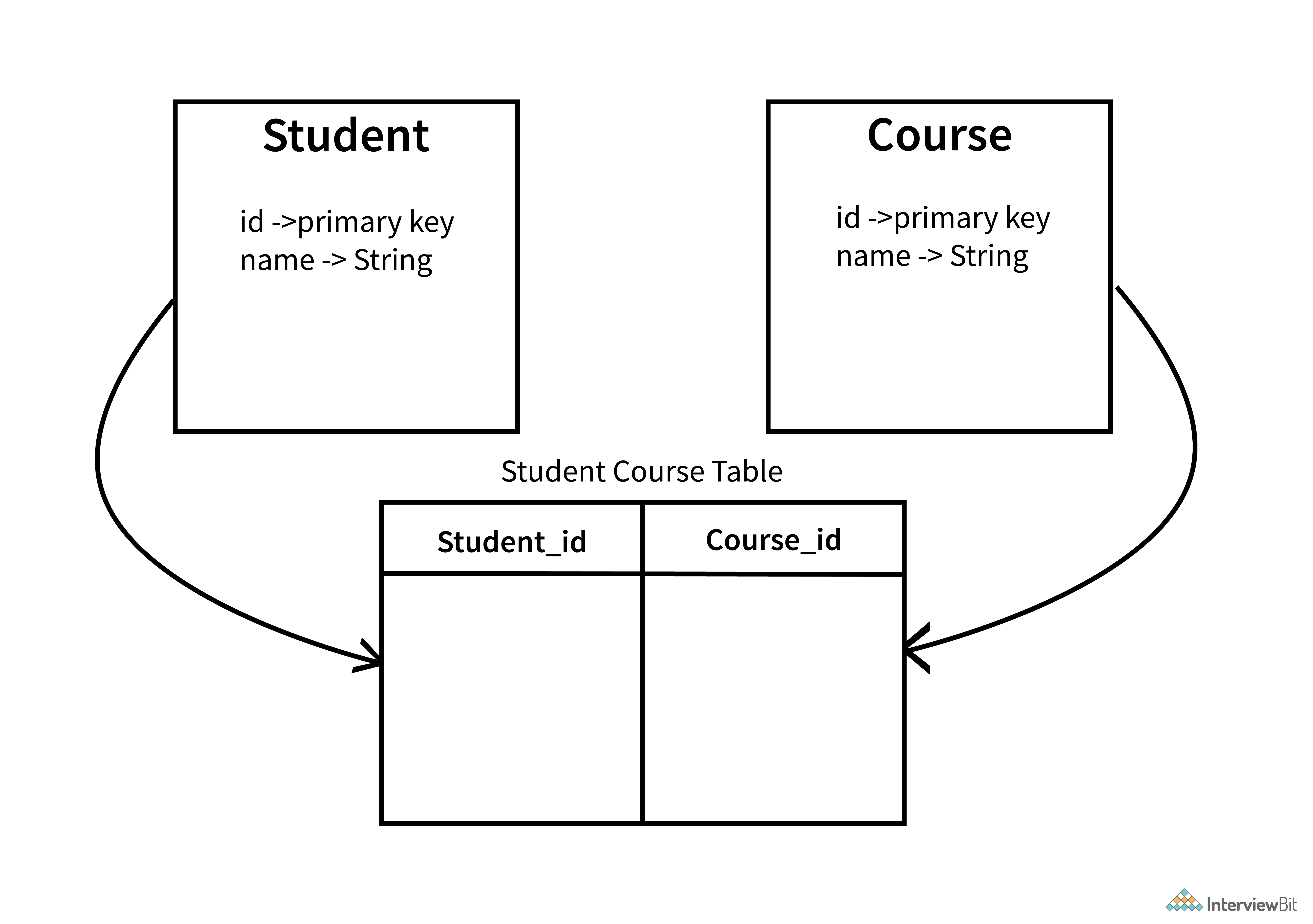


**3. Many-to-One (N:1) Association:**

* In a many-to-one association, multiple records in one entity can be associated with a single record in another entity.
* Example: Multiple Order entities can be associated with a single Customer entity, as each order is placed by one customer.

**4. Many-to-Many (N:N) Association:**

* In a many-to-many association, multiple records in one entity can be associated with multiple records in another entity.
* Example: A Student entity can be associated with multiple Course entities, and each course can have multiple students enrolled in it.



**5. Bidirectional vs. Unidirectional:**

* Associations can be bidirectional or unidirectional based on whether the association is navigable from both sides or only from one side.
* Bidirectional associations allow navigation and querying from both entities, while unidirectional associations allow navigation only from one entity to the other.

**Que: Collection mapping can be done using One-to-One and Many-to-One Associations. What do you think?**

**Ans:**

Collection mapping in Hibernate is typically done using One-to-Many and Many-to-Many associations rather than One-to-One and Many-to-One associations. Let's discuss why

Example: A Department entity can have multiple Employee entities associated with it, making it suitable for a collection mapping.

A Student entity can be associated with multiple Course entities, and each course can have multiple students, making it suitable for a collection mapping.

**Que: Can we declare the Entity class final?**

**Ans:** It's not recommended to declare an entity class as final in Hibernate because it can hinder proxy generation, inheritance mapping, dynamic proxies, persistence context management, and extensibility. This restriction can lead to limitations and compatibility issues with Hibernate's features and optimizations.

**JPA Interview Question:**

**Que: What are JPA?**

**Ans:** JPA stands for Java Persistence API. It is a standard Java specification for object-relational mapping (ORM) frameworks that allows Java developers to interact with relational databases using object-oriented principles. JPA provides a set of interfaces, annotations, and APIs for managing persistent data in Java applications. Here are key points about JPA:

**1. ORM Framework:**

* JPA is an ORM framework that abstracts the mapping between Java objects (entities) and relational database tables.
* It allows developers to work with entities and perform CRUD (Create, Read, Update, Delete) operations on them without writing SQL queries explicitly.

0

**2. Standard Specification:**

* JPA is part of the Java EE (Enterprise Edition) specification and is maintained by the Java Community Process (JCP).
* It provides a standard way for Java developers to perform database operations using a common set of APIs and annotations.

**3. Persistence Context:**

* JPA introduces the concept of a persistence context, which is a collection of managed entities and their associated states.
* The persistence context tracks changes made to entities and synchronizes these changes with the database during transaction commit.

**4.Query Language:**

* JPA introduces the Java Persistence Query Language (JPQL), which is a database-independent query language similar to SQL.
* JPQL allows developers to write queries using entity attributes and relationships, providing a more object-oriented approach to querying.

**5. Vendor Implementations:**

* JPA is a specification, and it requires an implementation to be used in applications.
* Several ORM frameworks provide JPA implementations, such as Hibernate, EclipseLink, OpenJPA, and DataNucleus.

**Que: What is the difference between Hibernate and JPA?**

**Ans**:

* Hibernate is an ORM framework developed by Red Hat, while JPA (Java Persistence API) is a Java specification for ORM.
* Hibernate implements the JPA specification, providing additional features beyond JPA.
* Hibernate-specific APIs and annotations can reduce portability, as they tie code to the Hibernate implementation.
* JPA promotes portability, interoperability, and vendor neutrality by defining a standard set of APIs and annotations.
* Applications developed using JPA can be more easily migrated to other JPA-compliant ORM frameworks.

*In short, Hibernate is a tool that follows the rules set by JPA. When you use Hibernate, you are essentially using JPA (because Hibernate implements JPA), but you also have access to additional features provided by Hibernate itself.*

*In other words, JPA provides a standard set of interfaces and annotations for ORM, while Hibernate is a concrete implementation of those interfaces and annotations.*

**Que: What is ORM Framework and how is JPA related to that?**

**Ans**:

* An ORM (Object-Relational Mapping) framework automates the mapping between objects in a programming language (like Java) and data stored in databases.
* JPA (Java Persistence API) is a set of specifications and APIs that define how ORM should be implemented in Java applications.
* ORM frameworks (such as Hibernate, EclipseLink) implement JPA specifications, providing standardized ways to work with databases in Java while also offering additional features and optimizations.
* Using an ORM framework with JPA allows developers to work with objects in code, perform CRUD operations, write queries, and manage transactions in a more intuitive and standardized manner, bridging the gap between object-oriented programming and relational databases.

**Que: How does JPA works?**

**Ans:** JPA (Java Persistence API) works by providing a standardized way for Java developers to interact with relational databases using object-oriented programming concepts. Here's an overview of how JPA works:

**1. Entity Classes:**

* Developers define entity classes in Java to represent persistent data objects. These classes are annotated with JPA annotations such as @Entity, @Table, @Column, @Id, etc., to specify their mapping to database tables, columns, and primary keys.

**2. Persistence Context:**

* JPA introduces the concept of a persistence context, which is a runtime environment that manages the lifecycle of entity instances. The persistence context is typically associated with a database transaction.
* When an entity is first accessed or loaded from the database, it becomes part of the persistence context. Changes made to managed entities within the persistence context are tracked and synchronized with the database during transaction commit.

**3. Entity Manager:**

* The Entity Manager is the central interface provided by JPA for performing CRUD (Create, Read, Update, Delete) operations and managing entity instances.
* Developers obtain an instance of Entity Manager from an Entity Manager Factory and use it to interact with the database.
* Entity Manager provides methods such as persist() to save new entities, find() to retrieve entities by their primary key, merge() to update entities, and remove() to delete entities.

**4.Transactional Operations:**

* JPA supports transactional operations, where multiple database operations are grouped into a single transaction. Transactions ensure data consistency and atomicity, allowing changes to be either committed or rolled back as a unit.
* Developers use annotations such as @Transactional (from Spring Framework) or @TransactionAttribute (from Java EE) to define transactional boundaries for methods that interact with the database.

**Que: What are entities in JPA? Explain the concept in detail.**

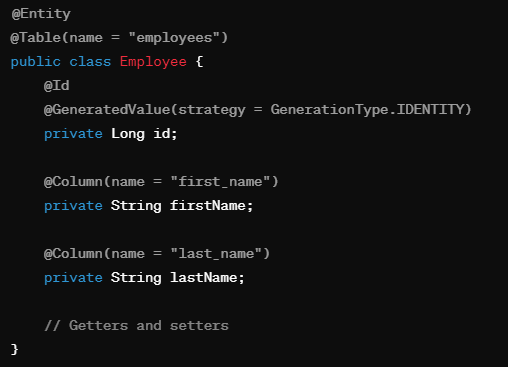
**Ans**: Entities in JPA (Java Persistence API) represent persistent data objects that are mapped to database tables. They are fundamental components of ORM (Object-Relational Mapping) and play a crucial role in defining the structure and behavior of data in a Java application. Here's a detailed explanation of entities in JPA:

**1. Definition:**

* An entity in JPA is a Java class that is annotated with @Entity or defined in XML configuration as an entity.
* Entities represent persistent data objects that are stored in a relational database.
* Each entity typically corresponds to a table in the database, with each instance of the entity representing a row in that table.

**2. Attributes:**

* Entities have attributes that map to columns in the corresponding database table.
* Attributes are defined using Java fields in the entity class, and they are annotated with @Column to specify the mapping to database columns.

****

**3. Primary Key:**

* Entities must have a primary key attribute annotated with @Id to uniquely identify each instance.
* The primary key can be a simple attribute (like id in the example above) or a composite key composed of multiple attributes.
* The @GeneratedValue annotation can be used to automatically generate unique primary key values.

**4. Relationships:**

* Entities can have relationships with other entities, such as one-to-one, one-to-many, many-to-one, and many-to-many relationships.
* Relationships are defined using annotations such as @OneToOne, @OneToMany, @ManyToOne, @ManyToMany, along with @JoinColumn and @JoinTable annotations to specify join columns and join tables.

**5. Lifecycle Callbacks:**

* Entities can define lifecycle callback methods using annotations like @PrePersist, @PostPersist, @PreUpdate, @PostUpdate, @PreRemove, and @PostRemove.
* These callback methods allow developers to execute custom logic before or after entity persistence operations (such as insert, update, delete) occur.

**6. Entity Manager:**

* The Entity Manager is a central interface provided by JPA for managing entity instances.
* Developers use the Entity Manager to perform CRUD operations (persist, find, merge, remove) on entities, as well as execute queries and manage transactions.

**7. Persistence Context**:

* Entities are managed within a persistence context, which is a runtime environment that tracks and manages entity instances.
* The persistence context ensures that changes made to managed entities are synchronized with the database during transaction commit.

**Que: What is the difference between Entity Manager, Entity Manager Factory and Persistance Context?**

**Ans:**

**EntityManager:**

* The EntityManager is a central interface in JPA for managing entity instances and performing CRUD (Create, Read, Update, Delete) operations on entities.
* It acts as a bridge between your Java application and the database, allowing you to interact with persistent entities using object-oriented principles.
* Key responsibilities of EntityManager include:
* Persisting new entities (persist())
* Retrieving entities by primary key or query (find(), createQuery())
* Updating managed entities (merge())
* Removing entities (remove())
* Executing database operations within transactions (beginTransaction(), commit(), rollback())
* EntityManager is typically obtained from an EntityManagerFactory.

**EntityManagerFactory:**

* The EntityManagerFactory is a factory class responsible for creating EntityManager instances.
* It is typically created once during application startup and shared across the application.
* EntityManagerFactory is configured with persistence unit settings, including database connection details, entity mappings, and other JPA properties.
* When you need to interact with the database, you obtain an EntityManager from the EntityManagerFactory.

**Persistence Context:**

* The Persistence Context is a runtime environment managed by EntityManager that tracks and manages entity instances during the lifecycle of a transaction.
* It acts as a first-level cache for managed entities, storing their state and tracking changes made to them.

*Entity manager overall manage krta hai pahle jis data se khelna hai uska instance banata hai... sara crud operation jo b krtna hota hai sab yahi krta hai...Java application aur database ke beech me bridge ka kam karta hai..*

*Entity Factory application chalu hote hi ban jata hai isme database ka seting credential hota hai setup karta hai connection extablich karta hai ... me hi entity manager ka jab requirement hota hai tabn yahi se ata hai wo... yaha ban ke rakha rahta hai jab jab jaise jaise use ho leta jata hai..*

*Persistnace context runtime environment hota hai jo entity manager ke andr create hota hai aur wahi usko manage karta hai...jo ki transaction ke sare reqcords ka track rakta hai..*

**Example:**

**Employee Entity:**

* Imagine you have a class called Employee that represents employees in a company.
* Each employee has attributes like a first name, last name, and an ID (which uniquely identifies them).

**Persistence Unit Configuration:**

* Think of the persistence unit configuration as setting up a connection to a database where all employee data will be stored.
* You specify details like the database URL, username, password, and driver information in a configuration file.

**EntityManagerFactory:**

* The EntityManagerFactory is like a factory that creates special managers called EntityManagers.
* This factory knows how to talk to the database based on the configuration you provided.

**EntityManager:**

* An EntityManager is like a manager in charge of handling employees (entities) and their information.
* It can create, find, update, or remove employees from the database.

**Persistence Context:**

* Think of the Persistence Context as a workspace where the EntityManager keeps track of employees and their changes.
* When you make changes to an employee (like updating their name), the EntityManager remembers these changes until you tell it to save them to the database.

**Example in Layman's Terms:**

* Imagine you want to add a new employee named John Doe to the company's database.
* You use the EntityManagerFactory to create an EntityManager.
* The EntityManager creates a workspace (Persistence Context) and manages John Doe's information within that workspace.
* You tell the EntityManager to add John Doe to the workspace.
* When you're ready, you ask the EntityManager to save John Doe's information to the database (committing the changes).
* The EntityManager uses the Persistence Context to track the changes and ensure they are saved correctly.

**Que: Explain in detail the JPA application life cycle?**

**Ans**: Stages are:

1. **Entity Class Creation:** The first stage in the lifecycle of a JPA application is the creation of entity classes. Entity classes are Java classes that represent database tables and have properties that correspond to columns in those tables.
2. **Entity Mapping:** The next stage is entity mapping, which involves defining the mapping between the entity classes and the database tables. This is typically done using annotations or XML configuration files, and it specifies how the properties of the entity classes correspond to the columns in the database tables.
3. **Persistence Unit Creation:** The third stage is the creation of a Persistence Unit, which is a logical grouping of one or more entity classes and their associated metadata. This is typically done using a persistence.xml file, which specifies the database connection details, the list of entity classes to be managed, and any additional configuration options.
4. **EntityManagerFactory Creation:** The next stage is the creation of an EntityManagerFactory, which is responsible for creating EntityManager instances. The EntityManagerFactory is typically created once at the start of the application and is used to create EntityManager instances throughout the application.
5. **EntityManager Creation**: The next stage is the creation of an EntityManager, which provides the primary interface for interacting with the Persistence Context. The EntityManager is responsible for managing the lifecycle of entity objects, executing queries, and performing CRUD operations on the database.
6. **Transaction** **Management**: The next stage is transaction management, which involves defining the boundaries of transactions and managing their lifecycle. Transactions are used to ensure data consistency and integrity, and they are typically managed using annotations or programmatic APIs.
7. **Entity** **Lifecycle** **Management**: The next stage is entity lifecycle management, which involves managing the lifecycle of entity objects within the Persistence Context. Entity objects can be in one of several states, including New, Managed, Detached, and Removed, and their state can be changed using the EntityManager API.
8. **Query** **Execution**: The final stage is query execution, which involves executing JPQL queries to retrieve data from the database. JPQL is a query language that is similar to SQL but is specific to JPA.

**Que: What are the advantages of using JPA over JDBC?**

**Ans**: The advantages of using JPA over JDBC are:

* Object-Relational Mapping (ORM): Simplifies data handling by mapping Java objects to database tables.
* Productivity: Reduces boilerplate code and allows developers to focus on business logic.
* Database Independence: Makes applications portable across different databases.
* Query Language: Offers JPQL for flexible and database-independent querying.
* Automatic Dirty Checking: Tracks changes to entities and synchronizes them with the database automatically.
* Caching and Performance: Provides caching mechanisms for faster data retrieval and improved performance.
* Transaction Management: Simplifies transaction handling with built-in APIs and annotations.
* Integration: Seamlessly integrates with Java EE and Spring frameworks for enterprise-level features and functionality.

**Que: Describe in detail about the Persistence Unit in JPA?**

**Ans:** In JPA (Java Persistence API), a Persistence Unit is a fundamental concept that represents a logical grouping of entity classes, entity mappings, and other configuration settings related to persistence in a Java application. It is defined in a persistence.xml file (for Java EE applications) or application properties (for Spring Boot applications) and serves as a configuration unit for JPA providers like Hibernate, EclipseLink, etc.

**Que: What is Spring Data JPA?**

**Ans**: Spring Data JPA is a part of the larger Spring Data project, which aims to simplify and streamline data access in Java applications by providing a higher-level abstraction over traditional data access technologies. Specifically, Spring Data JPA focuses on integrating JPA (Java Persistence API) with the Spring Framework to offer powerful features and reduce the amount of boilerplate code required for data access tasks.

JPA aur spring ka integrating hai aur kch ni sab same hai.

Here are the key aspects and features of Spring Data JPA:

**1. Integration with JPA:**

* Spring Data JPA integrates seamlessly with JPA, which is a Java specification for ORM (Object-Relational Mapping). It leverages JPA's features for mapping Java objects to database tables and executing database queries.

**2. Repository Abstraction:**

* Spring Data JPA provides a repository abstraction that allows developers to define repositories for JPA entities using interfaces. These repositories handle common data access operations like CRUD (Create, Read, Update, Delete) operations without requiring explicit implementation.

**3. Automatic Query Generation:**

* By following naming conventions and method signatures in repository interfaces, Spring Data JPA automatically generates queries for common data access methods. For example, a method named findByLastName(String lastName) in a repository interface generates a query to find entities by their last name.

**4. Custom Query Methods:**

* Developers can define custom query methods in repository interfaces using @Query annotations or query derivation. This allows for more complex and customized database queries while still benefiting from Spring Data JPA's repository abstraction.

**5. Pagination and Sorting:**

* Spring Data JPA provides built-in support for pagination and sorting of query results, making it easy to handle large datasets efficiently.

**6. Transaction Management:**

* Spring Data JPA integrates with Spring's transaction management capabilities, allowing developers to manage transactions declaratively using annotations like @Transactional.

**Que: What is JPA repository?**

**Ans:** JPA Repository is a feature provided by Spring Data JPA, which simplifies data access and manipulation in Java applications that use JPA (Java Persistence API). It is a part of the Spring Data project, which aims to abstract away the complexities of working with data access layers.

In simpler terms, a JPA Repository is an interface provided by Spring Data JPA that extends the basic CRUD (Create, Read, Update, Delete) operations commonly used in data access layers. It allows developers to perform database operations on JPA entities without writing boilerplate code for common tasks like saving, retrieving, updating, or deleting data.

It provides a set of methods for performing common operations on entities, such as save, delete, findAll, and findById. In addition to these methods, it also allows you to define custom query methods using the @Query annotation.

**Que: Difference between JPA Repository and CRUD Repository? Explain with the help of an example.**

**Ans:**

**JPA Repository:**

* JpaRepository is an interface provided by Spring Data JPA specifically for JPA (Java Persistence API) implementations like Hibernate, EclipseLink, etc.
* It extends the PagingAndSortingRepository interface and adds additional JPA-specific methods for working with JPA entities.
* JPA Repository provides methods for common CRUD (Create, Read, Update, Delete) operations as well as custom query methods.

**CrudRepository:**

* CrudRepository is a more generic interface provided by Spring Data that can work with any data store, not just JPA.
* It provides basic CRUD operations (save, findById, findAll, delete, etc.) for entities without any specific support for JPA features like paging, sorting, or custom queries.
* CrudRepository is suitable for simple data access scenarios where advanced features of JPA are not required.

**Que: What is a Named Query in JPA? How is it used? And what are the benefits of using this?**

**Ans**: In JPA, a named query is a pre-defined query that is given a name and can be used in multiple places in an application. It is defined in the entity class using the @NamedQuery annotation and can be used to retrieve entities based on specific criteria.

Benefits of Named Queries:

* **Code Readability and Maintainability**: Named Queries improve code readability by separating query logic from Java code, making it easier to understand and maintain.
* **Reuse of Queries:** You can reuse named queries across different parts of your application without duplicating query strings, promoting code reusability.
* **Compile-Time Checking:** Named Queries are checked at compile time, catching errors such as typos or incorrect syntax early in the development process.
* **Performance Optimization:** Some JPA providers optimize named queries during application startup, leading to improved performance for frequently executed queries.

**Que: What are the various query methods in JPA to retrieve data from the database? List some of the most used methods.**

**Ans:**

**Basic Query Methods:**

* findById(ID id): Retrieves an entity by its primary key.
* findAll(): Retrieves all entities of a specific type.
* count(): Counts the number of entities of a specific type.
* deleteById(ID id): Deletes an entity by its primary key.
* existsById(ID id): Checks if an entity with the given primary key exists.
* @Query("SELECT e FROM Entity e WHERE condition"): Executes a custom JPQL query.

**Que: What is the difference between EntityManager.find() and EntityManager.getReference() methods in JPA?**

**Ans:**

**EntityManager.find():**

* The find() method is used to retrieve an entity by its primary key or identifier.
* Syntax: entityManager.find(EntityClass.class, primaryKey)
* When you call find(), JPA immediately queries the database and fetches the entity if it exists.
* If the entity is not found in the database, find() returns null.

**EntityManager.getReference():**

* The getReference() method is used to obtain a reference (proxy) to an entity without actually loading its data immediately.
* Syntax: entityManager.getReference(EntityClass.class, primaryKey)
* getReference() returns a proxy object that acts as a placeholder for the actual entity.
* The actual database query to fetch the entity's data is deferred until the proxy object is accessed or a method is invoked on it.
* If the entity does not exist in the database and you try to access its data, a javax.persistence.EntityNotFoundException may be thrown.

**Que: What are all annotation used in Spring Data JPA?**

**Ans:** Spring Data JPA provides several annotations that help in configuring and customizing data access operations. Some of the most commonly used annotations in Spring Data JPA are as follows:

**1. @Entity:**

* Marks a Java class as an entity that can be mapped to a database table.

**2. @Table:**

* Specifies the name of the database table associated with an entity.

***@Entity***

***@Table(name = "users")***

***public class User {***

***// Entity attributes, getters, and setters***

***}***

**3. @Id:**

* Marks a field as the primary key of an entity.

**4. @GeneratedValue:**

* Specifies the strategy for generating primary key values.

***@GeneratedValue(strategy = GenerationType.IDENTITY)***

***private Long id;***

**5. @Column:**

* Specifies the mapping of an entity attribute to a database column.

***@Column(name = "username")***

***private String username;***

**6. @OneToMany and @ManyToOne:**

* Defines a one-to-many or many-to-one relationship between entities.

**7. @Query:**

* Defines a custom JPQL (Java Persistence Query Language) query or a native SQL query.

***@Query("SELECT u FROM User u WHERE u.username = :username")***

***User findByUsername(@Param("username") String username);***

**8. @NamedQuery and @NamedNativeQuery:**

* Declares a named JPQL or native SQL query for an entity class.

**Que: What types of identifier generation does JPA support?**

**Ans**: Generation strategies with their corresponding GenerationType:

1. **AUTO (GenerationType.AUTO):**

* Automatically selects the appropriate strategy based on the database capabilities.

1. **IDENTITY (GenerationType.IDENTITY):**

* Uses database auto-increment columns for ID generation (e.g., MySQL).

1. **SEQUENCE (GenerationType.SEQUENCE):**

* Utilizes database sequences for ID generation (e.g., Oracle).

1. **TABLE (GenerationType.TABLE):**

* Generates IDs using a dedicated database table.

1. **SEQUENCE or IDENTITY (GenerationType.AUTO in Hibernate):**

* Tries sequence first, then falls back to identity (Hibernate-specific).

1. **CUSTOM (GenerationType.IDENTITY with custom generator):**

* Allows developers to implement custom ID generation logic using custom generators.

**Que: What is Optimistic Locking? How to handle it in JPA?**

**Ans:** Optimistic locking is a concurrency control mechanism used in databases to handle concurrent access to data by multiple users or processes. In optimistic locking, the idea is to assume that conflicts between concurrent transactions are rare, so instead of locking data upfront, the system allows concurrent access and checks for conflicts only at the time of updating data.

Here's how it works:

**1. Version-Based Optimistic Locking:**

* In version-based optimistic locking, each entity that requires optimistic locking has a version attribute annotated with @Version.
* The version attribute is typically an integer or timestamp that gets updated automatically by JPA when an entity is modified.
* When an entity is fetched from the database, JPA tracks its version.
* When an update operation is performed on the entity, JPA checks if the version in the database matches the version of the entity being updated.
* If the versions match, the update is allowed, and the version in the database is incremented.
* If the versions don't match (indicating that another transaction has modified the entity in the meantime), an optimistic locking exception (OptimisticLockException) is thrown.

**2. Optimistic Locking Annotations:**

* JPA provides annotations to configure optimistic locking behavior:
* @Version: Marks a field as the version attribute for optimistic locking.
* @OptimisticLocking: Specifies the type of optimistic locking (VERSION or NONE).
* @Versioned: A shortcut annotation that combines @Version and @OptimisticLocking.
* These annotations provide flexibility in defining how entities are managed and locked optimistically.

**Que: What is the purpose of @Transactional?**

**Ans**: The @Transactional annotation in Spring Framework is used to indicate that a method should be executed within a transaction context. Transactions are used to ensure data consistency and integrity by grouping related operations together and either committing all of them or rolling back if an error occurs. Here's the purpose and functionality of @Transactional:

When you annotate a method with @Transactional, Spring creates a transactional boundary around that method. This means that all operations performed within the method will be considered part of the same transaction.

***@Service***

***public class UserService {***

***@Autowired***

***private UserRepository userRepository;***

***@Transactional***

***public void updateUser(User user) {***

***// Perform update operations on the user entity***

***userRepository.save(user);***

***}***

***// Other transactional methods***

*}*

**Que: Disadvantages of Hibernate and JPA?**

**Ans**: Certainly! Here are the disadvantages of Hibernate and JPA summarized in short:

* Complexity: Introduce additional complexity due to ORM abstraction.
* Performance Overhead: May have performance overhead compared to raw SQL or JDBC.
* Learning Curve: Steep learning curve for ORM concepts and configurations.
* Automatic Query Generation: Generated queries may not always be optimal.
* Handling Complex Queries: Challenging for complex SQL queries.
* Database Vendor Specifics: Differences in behavior across database vendors.
* Caching Challenges: Managing caching can be complex and lead to issues.

**Que: Mention the steps to connect the Spring Boot application to a database using JDBC.**

**Ans:** To connect an external database like MySQL or Oracle to a Spring Boot application using JDBC, we need to follow below steps:

* Add the dependency for the JDBC driver of the database.
* Create an application.properties file.
* Configure the database connection properties.
* Create a JdbcTemplate bean.
* Use the JdbcTemplate bean to execute SQL queries and statements.

**Que: Explain Spring Data and What is Data JPA?**

**Ans:**

**Spring Data:**

* Spring Data is a collection of modules that provide enhanced support for data access operations in Spring-based applications.
* It offers abstractions and APIs to work with different data stores, including relational databases (e.g., MySQL, PostgreSQL), NoSQL databases (e.g., MongoDB, Redis), and cloud-based storage services.

**Spring Data JPA:**

* Spring Data JPA is a module of Spring Data that provides additional functionality and abstractions specifically for working with relational databases using the Java Persistence API (JPA).

**Que: How is Hibernate chosen as the default implementation for JPA without any configuration?**

**Ans:** Spring Boot automatically configures Hibernate as the default JPA implementation when we add the spring-boot-starter-data-jpa dependency to our project. This dependency includes the Hibernate JAR file as well as the Spring Boot auto-configuration for JPA.

**Spring REST API**

**Que: What do you understand by RESTful Web Services?**

**Ans:** RESTful Web Services, or Representational State Transfer (REST) services, are an architectural style for designing networked applications. REST is based on a set of principles that define how resources are represented and accessed over the web. Here's what you need to know about RESTful Web Services:

**Resource-Based:** RESTful services are centered around resources, which can be any entity or object that needs to be accessed, such as users, products, or documents. Each resource is uniquely identified by a URI (Uniform Resource Identifier).

**HTTP Methods:** RESTful services use standard HTTP methods (GET, POST, PUT, DELETE, etc.) to perform operations on resources. For example:

* GET: Retrieve data from a server.
* POST: Submit data to be processed by a server.
* PUT: Update or replace a resource on a server.
* PATCH: Partially modify a resource on a server.
* DELETE: Remove a resource from a server.
* HEAD: Retrieve headers of a resource without its body.
* OPTIONS: Retrieve communication options for a resource.
* TRACE: Echo back the received request for debugging.
* CONNECT: Establish a secure tunnel through proxies.

**Stateless Communication:** REST is stateless, meaning each request from the client to the server contains all the information needed to process the request. The server does not store any client state between requests, improving scalability and reliability.

**Que: What is a REST Resource?**

**Ans:** A REST resource is a key concept in RESTful architecture, representing a piece of information or an object that can be accessed and manipulated through a RESTful API.

They can either be represented as text files, HTML pages, images, or any other dynamic data.

Here are the key characteristics and components of a REST resource:

**URI (Uniform Resource Identifier):**

* Each REST resource is uniquely identified by a URI, which serves as its address or location on the web.
* Example: https://example.com/api/users/123

**Representation:**

* A resource can have different representations, such as JSON, XML, HTML, or plain text.
* Clients can request different representations based on their preferences or needs (content negotiation).

**Que: What is URI?**

**Ans:** URI stands for Uniform Resource Identifier. It is a string of characters that identifies a particular resource, either on the Internet or within a local network. URIs are used to uniquely identify and locate resources such as web pages, files, images, API endpoints, and more.

**Components of a URI:**

* **Scheme:** Indicates the protocol or type of resource (e.g., HTTP, HTTPS, FTP).
* **Authority**: Typically includes the domain name or IP address of the server hosting the resource.
* **Path**: Specifies the specific location or name of the resource on the server.
* **Query** **String**: Contains additional parameters or data for the resource, usually in key-value pairs.
* **Fragment** **Identifier**: Optional component used to identify a specific portion or section of the resource (e.g., in HTML anchors).

**Types** **of** **URIs**:

* **URL (Uniform Resource Locator**): A type of URI that specifies the web address of a resource and how to access it (e.g., https://example.com/page).
* **URN (Uniform Resource Name):** A type of URI that uniquely identifies a resource by its name or identifier, independent of location or access method (e.g., urn:isbn:0451450523).
* **Example of a URI:**

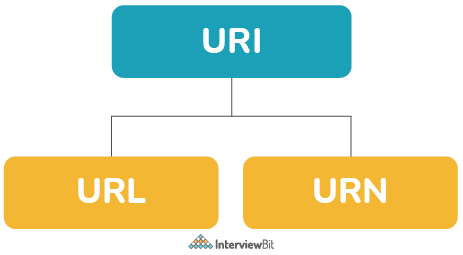
URI: https://example.com/api/users?id=123

Scheme: https

Authority: example.com

Path: /api/users

Query String: id=123



**Que: What are the features of RESTful Web Services?**

**Ans:** Here are the features of RESTful Web Services:

**1.Resource-Based:**

RESTful services are resource-centric, where each resource is uniquely identified by a URI (Uniform Resource Identifier).

**2. Stateless Communication:**

RESTful services are stateless, meaning that each request from the client to the server contains all the information needed to process the request.

**3. Representation:**

Resources in RESTful services can have multiple representations (e.g., JSON, XML, HTML) based on client preferences or content negotiation.

**4. Client-Server Architecture:**

REST follows a client-server architecture, where clients initiate requests and servers respond with data or actions.

**5. State Transfer:**

REST emphasizes the transfer of state between client and server through representations of resources.

**6. Cacheability:**

RESTful responses can be cached to improve performance and reduce server load.

Cache control headers in HTTP responses specify whether a response can be cached and for how long.

**Que: What are HTTP Status codes?**

**Ans:** HTTP status codes are standardized codes used by web servers to indicate the outcome of an HTTP request made by a client (such as a web browser or an API client).

**1. Informational (1xx):**

These status codes indicate that the server has received the request and is processing it.

Example: 100 Continue (Server acknowledges the initial part of the request and expects the client to continue sending the rest of the request).

**2. Success (2xx):**

These status codes indicate that the request was successfully received, understood, and processed by the server.

Example: 200 OK (Standard response for successful HTTP requests).

**3. Redirection (3xx):**

These status codes indicate that further action needs to be taken by the client to complete the request.

Example: 301 Moved Permanently (The requested resource has been permanently moved to a new location, and the client should update its URL).

**4. Client Error (4xx):**

These status codes indicate that there was an error or issue with the client's request.

Example: 404 Not Found (The requested resource could not be found on the server).

**5. Server Error (5xx):**

These status codes indicate that there was an error or issue on the server side while processing the request.

Example: 500 Internal Server Error (A generic error message indicating that something went wrong on the server).

Some commonly used HTTP status codes include:

**200 OK**: Successful request.

**201 Created:** Resource successfully created**.**

**400 Bad Request:** Invalid request from the client.

**401 Unauthorized:** Authentication required or credentials invalid.

**403 Forbidden:** Client does not have permission to access the resource**.**

**404 Not Found:** Resource not found on the server.

**500 Internal Server Error:** Server encountered an unexpected condition.

**Que: Define Messaging in terms of RESTful web services.**

**Ans:** The technique of sending a message from the REST client to the REST server in the form of an HTTP request and the server responding back with the response as HTTP Response is called Messaging. The messages contained constitute the data and the metadata about the message.

**Que: What makes REST services to be easily scalable?**

**Ans**: REST services follow the concept of statelessness which essentially means no storing of any data across the requests on the server. This makes it easier to scale horizontally because the servers need not communicate much with each other while serving requests.

**Que: What is Payload in terms of RESTful web services?**

**Ans**: Payload refers to the data passes in the request body. It is not the same as the request parameters. The payload can be sent only in POST methods as part of the request body.

**Que: Should we make the resources thread safe explicitly if they are made to share across multiple clients?**

**Ans:** There is no need to explicitly making the resources thread-safe because, upon every request, new resource instances are created which makes them thread-safe by default.

**Que: Is it possible to send payload in the GET and DELETE methods?**

**Ans:** No, the payload is not the same as the request parameters. Hence, it is not possible to send payload data in these methods.

**17. How can you test RESTful Web Services?**

**Ans:** RESTful web services can be tested using various tools like Postman, Swagger, etc. Postman provides a lot of features like sending requests to endpoints and show the response which can be converted to JSON or XML and also provides features to inspect request parameters like headers, query parameters, and also the response headers. Swagger also provides similar features like Postman and it provides the facility of documentation of the endpoints too. We can also use tools like Jmeter for performance and load testing of APIs.

**Que: What are Spring REST API?**

**Ans:** Spring REST API refers to building RESTful web services using the Spring Framework, particularly the Spring MVC module. Spring MVC provides powerful features and annotations to create RESTful APIs easily.

Here are the key components and features of Spring REST API:

**1. Controller Layer:**

* Controllers in Spring MVC are responsible for handling incoming HTTP requests and generating HTTP responses.
* Spring MVC provides annotations like @RestController and @RequestMapping to create RESTful controllers.

**2. Mapping Requests to Methods:**

* Spring MVC maps incoming HTTP requests to specific controller methods based on annotations such as @GetMapping, @PostMapping, @PutMapping, @DeleteMapping, etc.
* These annotations allow developers to define methods for handling different HTTP methods (GET, POST, PUT, DELETE) on specific URIs.

**3. Data Representation:**

* Spring REST API supports various data representation formats such as JSON, XML, and others.
* It uses message converters to convert Java objects to JSON/XML and vice versa, allowing seamless data exchange between clients and servers.

**4. Response Handling:**

* Controllers return data using ResponseEntity for more control over HTTP status codes, headers, and response bodies.
* ResponseEntity allows customization of response metadata, content type, and error handling.

**5. Exception Handling:**

* Spring REST API provides robust exception handling mechanisms using @ExceptionHandler and @ControllerAdvice annotations.
* Developers can define global exception handlers to handle various types of exceptions and return appropriate error responses.

**6. Validation:**

* Spring MVC supports validation of request data using JSR-303/JSR-380 annotations such as @Valid and @Validated.
* It integrates with Bean Validation API to validate request parameters, request bodies, and form inputs.

**7. Content Negotiation:**

* Spring REST API supports content negotiation to produce/consume different representations of resources based on client preferences.
* Clients can specify desired media types (JSON, XML, etc.) using Accept and Content-Type headers.

**8. Security:**

* Spring Security can be integrated with Spring REST API to add authentication, authorization, and other security features.
* It supports various authentication mechanisms like Basic Authentication, OAuth, JWT, etc.

**9. Documentation**:

* Spring REST API documentation can be generated using tools like Swagger or Springfox, providing API documentation, testing, and exploration capabilities.

**Que: What are End points used in Spring REST API?**

**Ans:** In a Spring REST API, endpoints are used to define the URLs that clients can access to interact with the API. These endpoints map HTTP requests to specific controller methods in the application, allowing clients to perform various operations on resources. Here are the commonly used endpoints in a Spring REST API:

**GET Endpoint:**

* Used for retrieving data from the server.
* Example: GET /api/users - Retrieve all users.

**POST Endpoint:**

* Used for creating new resources on the server.
* Example: POST /api/users - Create a new user.

**PUT Endpoint:**

* Used for updating existing resources on the server.
* Example: PUT /api/users/{id} - Update user with ID {id}.

**PATCH Endpoint:**

* Similar to PUT but used for partial updates.
* Example: PATCH /api/users/{id} - Update specific fields of user with ID {id}.

**DELETE Endpoint:**

* Used for deleting resources from the server.
* Example: DELETE /api/users/{id} - Delete user with ID {id}.

**Custom Endpoints:**

* Custom endpoints can be defined for specific operations or actions.
* Example: GET /api/users/search?name=John - Search users by name.

**Subresource Endpoints:**

* Endpoints can represent subresources or related resources.
* Example: GET /api/users/{id}/posts - Retrieve posts of user with ID {id}.

**Pagination Endpoints:**

* Used for paginating large collections of resources.
* Example: GET /api/users?page=1&size=10 - Retrieve users with pagination.

**Sorting Endpoints:**

* Used for sorting resources based on certain criteria.
* Example: GET /api/users?sort=name,asc - Retrieve users sorted by name in ascending order.

**Filtering** **Endpoints**:

* Used for filtering resources based on specific conditions.
* Example: GET /api/users?active=true - Retrieve active users only.

**File Upload Endpoints:**

* Used for uploading files to the server.
* Example: POST /api/files/upload - Upload a file to the server.

**Authentication Endpoints:**

* Used for user authentication and authorization.
* Example: POST /api/auth/login - User login endpoint.

**Error Handling Endpoints:**

* Used for handling and returning error responses.
* Example: GET /api/error - Endpoint to trigger and handle an error.

**Que: What are Mapping Method in Spring REST API?**

**Ans**: In Spring REST, mapping methods are used to define how HTTP requests are mapped to specific controller methods based on the request URL, HTTP method, and other criteria. Spring provides several annotations for mapping methods in RESTful controllers. Here are the commonly used mapping annotations in Spring REST:

**@RequestMapping:**

* Used to map HTTP requests to controller methods based on URL patterns, HTTP methods, headers, and other request parameters.
* Example: @RequestMapping(value = "/api/users", method = RequestMethod.GET) - Maps GET requests to /api/users endpoint.

**@GetMapping:**

* Shortcut for mapping HTTP GET requests to controller methods.
* Example: @GetMapping("/api/users/{id}") - Maps GET requests to /api/users/{id} endpoint.

**@PostMapping:**

* Shortcut for mapping HTTP POST requests to controller methods.
* Example: @PostMapping("/api/users") - Maps POST requests to /api/users endpoint.

**@PutMapping:**

* Shortcut for mapping HTTP PUT requests to controller methods.
* Example: @PutMapping("/api/users/{id}") - Maps PUT requests to /api/users/{id} endpoint.

**@PatchMapping:**

* Shortcut for mapping HTTP PATCH requests to controller methods.
* Example: @PatchMapping("/api/users/{id}") - Maps PATCH requests to /api/users/{id} endpoint.

**@DeleteMapping:**

* Shortcut for mapping HTTP DELETE requests to controller methods.
* Example: @DeleteMapping("/api/users/{id}") - Maps DELETE requests to /api/users/{id} endpoint.

**Que: Define RESTTemplate in Spring?**

**Ans**: RestTemplate is a class provided by the Spring Framework that simplifies the process of making HTTP requests and handling RESTful web services. It encapsulates common HTTP operations like GET, POST, PUT, DELETE, etc., and provides methods for sending requests, receiving responses, and handling errors. Here's a detailed explanation of RestTemplate in Spring:

**HTTP Operations:**

* RestTemplate allows you to perform various HTTP operations such as GET, POST, PUT, DELETE, PATCH, etc., to interact with RESTful APIs.

**Request Building:**

* You can use RestTemplate to build HTTP requests by specifying the URL, request headers, request parameters, request body, etc.

**Response Handling:**

* After sending a request, RestTemplate handles the HTTP response, including status codes, response headers, and response body.

**Serialization and Deserialization:**

* RestTemplate can automatically serialize Java objects into JSON or XML format when sending requests and deserialize JSON/XML responses into Java objects.

**Error Handling:**

* RestTemplate provides error handling capabilities for handling exceptions, client errors (4xx), server errors (5xx), and other HTTP-related errors**.**

**Customization:**

* You can customize RestTemplate by adding interceptors, message converters, error handlers, authentication mechanisms, etc., to suit your application's requirements.

**Asynchronous Operations:**

* RestTemplate supports asynchronous HTTP operations using AsyncRestTemplate or by integrating with Spring's reactive programming model (WebClient).

**Integration with Spring Boot:**

* In Spring Boot applications, RestTemplate is auto-configured and can be easily injected into other Spring components using @Autowired.

**Que: What is the use of @RequestMapping?**

**Ans**: The @RequestMapping annotation in Spring MVC is used to map HTTP requests to specific handler methods in a controller class. It provides a flexible way to define URL patterns, HTTP methods, request parameters, headers, and other conditions for mapping requests to controller methods. Here's a detailed explanation of the @RequestMapping annotation and its uses:

**Mapping by URL Path:**

* You can use @RequestMapping to map requests based on the URL path.
* Example: @RequestMapping("/users") maps requests with the URL path /users to the annotated method.

**Mapping by HTTP Method:**

* @RequestMapping allows you to specify the HTTP method(s) that the handler method should respond to.
* Example: @RequestMapping(value = "/users", method = RequestMethod.GET) maps GET requests to the /users URL.

**Mapping by Consumes and Produces:**

* You can specify the content type (media type) of the request body that the handler method consumes and the response body that it produces.
* Example: @RequestMapping(value = "/users", method = RequestMethod.POST, consumes = MediaType.APPLICATION\_JSON\_VALUE) specifies that the handler method consumes JSON data in the request body.

**Mapping by Request Parameters:**

* @RequestMapping can map requests based on specific request parameters.
* Example: @RequestMapping(value = "/users", params = "id") maps requests to the /users URL only if they contain a parameter named id.

**Mapping by Headers:**

* You can map requests based on specific HTTP headers.
* Example: @RequestMapping(value = "/users", headers = "X-Requested-With=XMLHttpRequest") maps requests to the /users URL only if they have the X-Requested-With header set to XMLHttpRequest.

**Combining Conditions:**

* @RequestMapping allows you to combine multiple conditions (URL path, HTTP method, parameters, headers, etc.) using logical operators like AND (&&) and OR (||).
* Example: @RequestMapping(value = "/users", method = RequestMethod.GET, params = "id", headers = "Accept=application/json") combines URL path, GET method, presence of id parameter, and Accept header value to map requests.

**Flexible Mapping:**

* @RequestMapping provides flexibility in defining complex mappings for handler methods, allowing you to handle different types of requests and conditions.

**Que: Difference between Request Param and Request Mapping?**

**Ans:**

**Request Parameter:**

* Request parameters are values sent along with an HTTP request, typically in the URL query string or in the request body for POST requests.
* In Spring MVC, request parameters are accessed using the @RequestParam annotation in controller methods.
* Example: http://example.com/api/users?id=123&name=John - Here, id and name are request parameters.

**Request Mapping:**

* Request mapping, represented by the @RequestMapping annotation, is used to map HTTP requests to specific handler methods in a controller class.
* It defines the URL pattern(s) and other conditions under which a controller method should be invoked to handle incoming requests.
* Example: @RequestMapping("/users") - This maps requests with the URL path /users to the annotated method.

**Example:**

* Request parameter example: @GetMapping("/users") public ResponseEntity<User> getUserById(@RequestParam Long id) {...}
* Request mapping example: @RequestMapping(value = "/users", method = RequestMethod.GET) public ResponseEntity<List<User>> getAllUsers() {...}

**Que: What are the differences between the annotations @Controller and @RestController?**

**Ans**: The @Controller and @RestController annotations are both used in Spring MVC to define controller classes, but they have some key differences in their functionality and usage:

**@Controller Annotation:**

* The @Controller annotation is used to mark a class as a Spring MVC controller.
* It typically handles HTTP requests and generates HTTP responses, including rendering views (HTML pages) in traditional web applications.
* Controllers annotated with @Controller are suitable for applications that serve both web pages and RESTful APIs.

**@RestController Annotation:**

* The @RestController annotation is a specialized version of @Controller introduced in Spring 4 that is tailored for RESTful APIs.
* It combines @Controller and @ResponseBody annotations, which means that every method in a @RestController class returns data directly in the response body (e.g., JSON, XML) rather than rendering views.
* @RestController is ideal for building web services or APIs where the focus is on data exchange in formats like JSON or XML.

**Key Differences:**

**Response Handling:**

* @Controller methods return views (HTML pages) or model objects that are rendered by a view resolver.
* @RestController methods return data directly in the response body, typically in JSON or XML format, using the @ResponseBody annotation implicitly.

**Purpose:**

* @Controller is used for traditional web applications that generate HTML views and handle form submissions.
* @RestController is used for building RESTful APIs that handle data exchange in JSON, XML, or other formats.

**Usage:**

* @Controller is suitable when you need to render views or handle traditional web requests.
* @RestController is suitable when you want to expose RESTful services and return data directly in the response body.

**Que: What does the annotation @PathVariable do?**

**Ans:** @PathVariable annotation is used for passing the parameter with the URL that is required to get the data. The @PathVariable annotation in Spring MVC is used to extract values from the URI (Uniform Resource Identifier) template and map them to method parameters in a controller handler method.

**Usage:**

* @PathVariable is typically used in controller methods to capture dynamic segments of the URL path.
* It is placed before a method parameter in a controller method signature.

**Dynamic Path Segments:**

* In a RESTful API, URLs often have dynamic segments that represent resource identifiers, such as /users/{id}, where {id} is a dynamic path segment.
* @PathVariable allows you to extract the value of {id} and use it in your controller method.

**Mapping URI Template Variables:**

* When a request is made to a URL with dynamic path segments, Spring MVC matches the dynamic segments to the corresponding @PathVariable annotated method parameters based on their names.
* For example, if the URL is /users/{id} and {id} is mapped to a method parameter named id with @PathVariable, Spring will automatically bind the value of {id} to the id parameter in the method.

***@GetMapping("/users/{id}")***

***public ResponseEntity<User> getUserById(@PathVariable Long id) {***

***// Logic to fetch user data by ID***

***User user = userService.getUserById(id);***

***return ResponseEntity.ok(user);***

***}***

In this example, @PathVariable Long id captures the value of {id} from the URL /users/{id} and passes it to the getUserById method to fetch the user with the specified ID.

**Que: Is it necessary to keep Spring MVC in the classpath for developing RESTful web services?**

**Ans:** Yes. Spring MVC needs to be on the classpath of the application while developing RESTful web services using Spring. This is because, the Spring MVC provides the necessary annotations like @RestController, @RequestBody, @PathVariable, etc. Hence the spring-mvc.jar needs to be on the classpath or the corresponding Maven entry in the pom.xml.

**Que: Define HttpMessageConverter in terms of Spring REST?**

**Ans:** HttpMessageConverter is a strategic interface that specified a converter for conversion between HTTP Requests and responses. Spring REST uses the HttpMessageConverter for converting responses to various data formats like JSON, XML, etc. Spring makes use of the “Accept” header for determining the type of content the client expects. Based on this, Spring would find the registered message converter interface that is capable of this conversion.

**Spring MVC**

**Que: What is MVC?**

**Ans:** MVC refers to Model, View, and Controller. It is an architectural design pattern, which governs the application’s whole architecture. It is a kind of design pattern used for solving larger architectural problems.

MVC divides a software application into three parts that are:

* Model
* View
* Controller

**Que: What is Spring MVC?**

**Ans:**

* Spring MVC (Model-View-Controller) is a web framework provided by the Spring Framework for building dynamic web applications and to perform Rapid Application Development (RAD).
* It is built on the top of the Java Servlet API.
* It follows the MVC architectural pattern, which separates the application into three main components: Model, View, and Controller. Here's an overview of each component in Spring MVC:

**Model:**

* The Model represents the data and business logic of the application.
* It includes Java classes (POJOs) that hold application data, perform business operations, and interact with the database or other external systems.
* In Spring MVC, the Model is typically populated by the Controller and passed to the View for rendering.

**View:**

* The View is responsible for presenting the data from the Model to the user.
* It includes presentation components such as JSP (JavaServer Pages), Thymeleaf templates, HTML, or other view technologies.
* Views are used to generate the user interface and display data to the user.

**Controller:**

* The Controller acts as an intermediary between the user (via the web browser) and the application logic (Model and View).
* It handles incoming HTTP requests from the user, processes them, interacts with the Model to retrieve or update data, and determines the appropriate View to render the response.
* In Spring MVC, controllers are Java classes annotated with @Controller that define request mappings (URL mappings) to handle different types of requests.

**Que: What are the features of Spring MVC?**

**Ans:** Key features and functionalities of Spring MVC include:

* **Request Handling:** Controllers map incoming HTTP requests to specific handler methods based on request mappings.
* **Data** **Binding**: Spring MVC supports data binding between HTML form fields and Java objects, simplifying form submissions and data validation.
* **View** **Resolution**: Views are resolved using ViewResolver implementations, allowing for dynamic view rendering based on logical view names.
* **Interceptors** **and** **Filters**: Interceptors and filters can be used for cross-cutting concerns such as logging, authentication, authorization, and error handling.
* **Internationalization**: Spring MVC supports internationalization and localization for building multilingual applications.
* **Form** **Handling**: Spring MVC provides form tags and form validation mechanisms for handling HTML forms and form submissions.
* **RESTful** **Web** **Services**: While originally designed for traditional web applications, Spring MVC can also be used to develop RESTful APIs using annotations like @RestController and @RequestMapping.

**Que: What is Java Servlet API?**

**Ans:** The Java Servlet API is a set of interfaces and classes that allow developers to create web applications in Java. Servlets extend the functionality of a web server to handle HTTP requests and generate HTTP responses. Key aspects of the servlet API include:

* Servlet Interface: Defines methods for servlets to handle requests and responses.
* HttpServletRequest and HttpServletResponse: Interfaces for accessing request and response data.
* Servlet Lifecycle: Includes initialization, service, and destruction phases.
* Servlet Container: Manages servlets, routing requests, and handling servlet lifecycle.
* Servlet Mapping: Maps servlets to URL patterns for request handling.
* Session Management: Supports session tracking for maintaining state between requests.
* Filter and Listener: Provides mechanisms for pre-processing and post-processing tasks.

**Que: What is JSP?**

**Ans:**

* JSP is a technology used for developing dynamic web pages in Java.
* JSP pages are text-based documents that contain HTML or XML code along with embedded Java code (Java expressions, scriptlets, declarations) for dynamic content generation.
* JSP pages are translated into servlets by the JSP container (part of the servlet container) during runtime, and the resulting servlets handle the request processing.
* JSP simplifies the process of combining static content (HTML, XML) with dynamic content (Java code) to create interactive web pages.

**Que: What is the difference between Spring Boot and Spring MVC?**

**Ans:**

**Spring Boot:**

* Spring Boot is a framework designed to simplify the process of building and deploying production-ready Spring applications.
* It provides a streamlined development experience by offering auto-configuration, embedded servers (like Tomcat, Jetty), and starter dependencies.
* Spring Boot aims to reduce boilerplate configuration and make it easier to set up Spring-based applications, including web applications (using Spring MVC or other web frameworks), RESTful services, batch processing, and more.
* It includes features like health checks, metrics, logging, externalized configuration, and easy deployment options.

**Spring MVC:**

* Spring MVC is a web framework provided by the Spring Framework for building web applications using the Model-View-Controller (MVC) architectural pattern.
* It focuses specifically on web-related components such as controllers, views, and handlers for processing HTTP requests and generating HTTP responses.
* Spring MVC offers features like request mapping, data binding, validation, form handling, internationalization, and view resolution.
* It is part of the larger Spring ecosystem and can be used with other Spring modules and libraries to create full-fledged web applications.

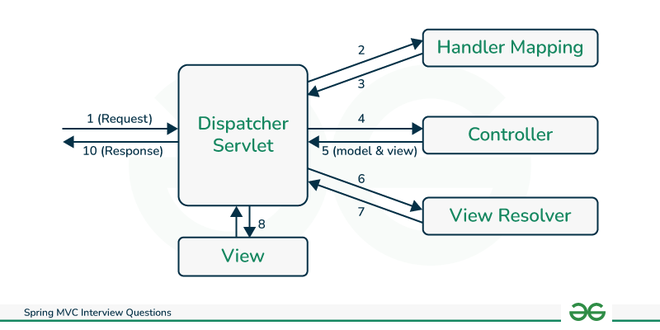
**In summary:**

* Spring Boot is a framework for simplifying the development and deployment of Spring applications, including web applications built using Spring MVC or other web frameworks.
* Spring MVC is a web framework within the Spring Framework that specifically focuses on handling web-related components and features for building MVC-based web applications.

**Que: Explain Spring MVC Architecture.**

**Ans:** Spring MVC Architectural Flow Diagram:

1. First, the request will come in through the browser and it will be received by Dispatcher Servlet, which will act as Front Controller.
2. Dispatcher Servlet will take the help of handler mapping and get to know the controller class name associated with the request.
3. After this, it will transfer the request to the controller, and then the controller will process the request by executing appropriate methods based on used GET or POST method.
4. And it will return the ModelAndView object back to the dispatcher servlet.
5. Now, the dispatcher servlet sends the model object to the view resolver in xml file to get the view page.
6. And finally, the dispatcher servlet will pass the model object to the view page to display the result.



**Que: What are DispatcherServlet, Handler Mapping and View Resolver?**

**Ans:**

**DispatcherServlet**

* The DispatcherServlet acts as a front controller in the Spring MVC framework.
* It receives all incoming HTTP requests from clients (web browsers, REST clients, etc.) and serves as the entry point for request processing in the application.
* DispatcherServlet is responsible for mapping incoming requests to specific handler methods (controllers) based on request mappings configured in the application.
* Request mappings are typically defined using annotations like @RequestMapping, @GetMapping, @PostMapping, etc., or through configuration files such as XML or Java-based configurations.
* Request Lifecycle Management:

DispatcherServlet manages the lifecycle of HTTP requests, including request parsing, data binding, validation, invoking controller methods, exception handling, and response rendering.

**Handler Mapping:**

* DispatcherServlet uses Handler Mapping components to determine which controller (handler) should process a specific request based on the request URL or other criteria.
* Spring provides various built-in handler mapping implementations, such as RequestMappingHandlerMapping, BeanNameUrlHandlerMapping, and SimpleUrlHandlerMapping.

**View Resolver**

* View Resolver resolves the logical view names returned by controllers into actual view implementations (JSP, Thymeleaf templates, etc.) for rendering.

There are different types of ViewResolver classes. Some of them are defined below:

1. **InternalResourceViewResolver:** It uses a prefix and suffix to convert a logical view name.
2. **ResourceBundleViewResolver:** It uses view beans inside property files to resolve view names.
3. XMLViewResolver: It also resolves view names in XML files to beans defined in the configuration file.

**Que: Explain the five most used annotations in Spring MVC Project.**

**Ans**: The five most commonly used annotations in a Spring MVC project:

**1. @Controller:**

* Used to annotate classes that handle HTTP requests in a Spring MVC application.
* Indicates that the annotated class serves as a controller component and contains request mapping methods.

**2. @RequestMapping:**

* Used to map HTTP requests to specific handler methods in a controller class.
* Specifies the URL path and HTTP method (GET, POST, PUT, DELETE, etc.) that the handler method should respond to.

***@Controller***

***@RequestMapping("/products")***

***public class ProductController {***

***@GetMapping("/list")***

***public String listProducts(Model model) {***

***// Method logic here***

***return "products";***

***}***

***}***

**3. @RequestParam:**

* It obtains the parameter from URL and Used to bind request parameters from the URL query string or form data to method parameters in a controller handler method.
* Specifies the name of the request parameter and provides optional attributes such as defaultValue and required.

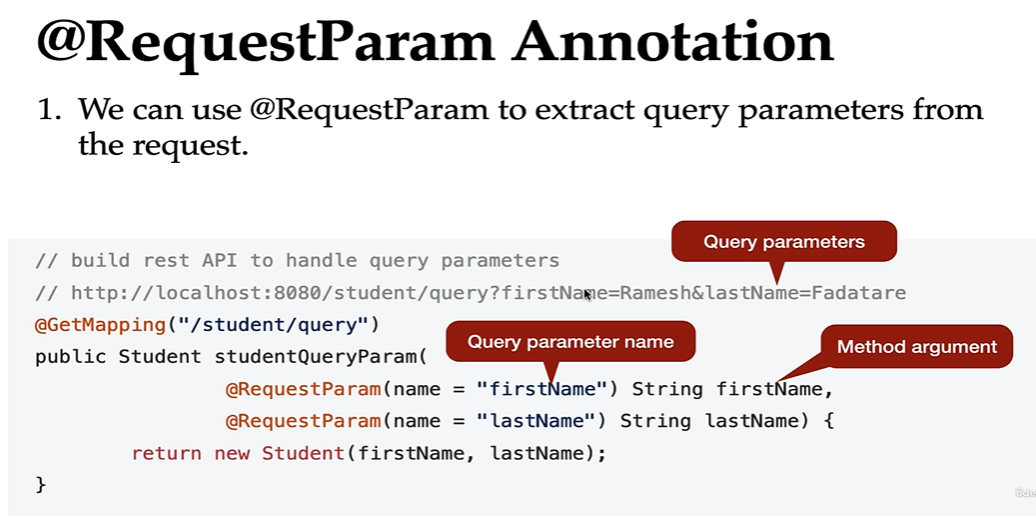
***@GetMapping("/product")***

***public String getProduct(@RequestParam("id") Long productId, Model model) {***

***// Method logic here***

***return "product";***

***}***

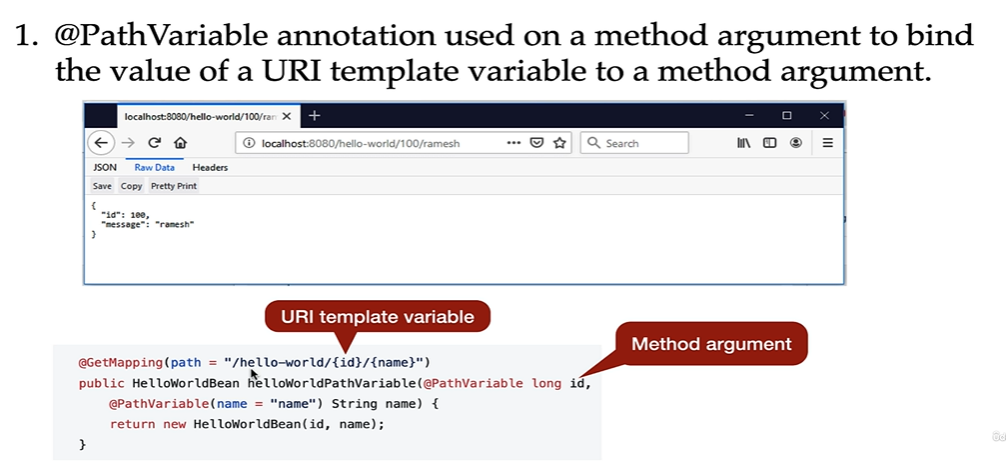
****

**4. @ResponseBody:**

* Used to indicate that the return value of a controller method should be serialized directly into the HTTP response body.
* Useful for building RESTful APIs that return JSON, XML, or other data formats directly to clients.

**5. @PathVariable:**

* Used to extract values from URL path variables and bind them to method parameters in a controller handler method.
* Allows for dynamic mapping of URL paths with variable segments.



**Que: What is WebApplicationContext in Spring MVC?**

**Ans:** In Spring MVC, the WebApplicationContext is a specialized ApplicationContext interface that is designed for web applications. It builds upon the functionality of the generic ApplicationContext interface provided by the Spring Framework but adds features and configuration specific to web environments.

Each servlet in the application (e.g., DispatcherServlet) can have its own WebApplicationContext instance, known as the servlet context, which manages beans and configurations specific to that servlet.

**Que: What is Servlet in Spring MVC?**

**Ans:** In Spring MVC, servlets are Java classes that handle HTTP requests and responses within the web application.

**Que: What is difference between a Servlet and Dispatcher servlet?**

**Ans:** Servlet and DispatcherServlet are both components of the Java Servlet API, but they serve different purposes in the context of web application development.

**Servlet:**

* A Servlet is a Java class that handles HTTP requests and generates HTTP responses. It is responsible for processing client requests, executing business logic, and producing HTML or other types of responses.

**DispatcherServlet:**

* The DispatcherServlet is a specialized servlet provided by the Spring MVC framework. It acts as a front controller that receives all incoming requests and delegates them to the appropriate controllers and handler methods based on URL mappings.

**Que: What is Request Dispatcher?**

**Ans:** The RequestDispatcher interface in Java Servlet API provides a way to forward a request from one servlet to another servlet or include the response from another servlet in the current response.

**forward():**

* The forward() method of the RequestDispatcher interface is used to forward the request from one servlet to another servlet or a resource (JSP, HTML file, etc.) on the server.

**include():**

* The include() method of the RequestDispatcher interface is used to include the response from another servlet or resource within the current response.

**Que: How to perform Validation in Spring MVC?**

**Ans:** Performing validation in Spring MVC involves several steps to ensure that user input data is valid and meets the specified criteria before processing it further.

**1. Add Validation Annotations:**

* Annotate the fields in your form-backing bean (model class) with validation annotations from the javax.validation.constraints package or custom validation annotations if needed.
* Example annotations include @NotNull, @Size, @Email, @Pattern, etc., to validate fields like required inputs, string length, email format, regular expressions, etc.

***public class User {***

***@NotNull***

***private String username;***

***@Size(min = 8, max = 20)***

***private String password;***

***@Email***

***private String email;***

***// Getters and setters***

***}***

**2. Enable Validation in Controller**:

* In your controller class, use @Valid or @Validated annotation along with BindingResult parameter to trigger validation on the form-backing bean when it's submitted.
* BindingResult captures validation errors and allows you to handle them in your controller.

**3. Display Validation Errors in View:**

* In your view (HTML form), use Thymeleaf, JSP, or other templating engines to display validation errors next to form fields.
* Use ${#fields.errors('fieldName')} to display validation errors for a specific field.

**Que: Life Cycle of a Servlet ?**

**Ans:** The servlet life cycle involves the following phases:

* **Loading and Instantiation:** Servlet container loads and instantiates the servlet when needed.
* **Initialization:** Servlet's init() method is called to initialize resources and configuration.
* **Request** **Handling**: Servlet's service() method processes client requests and generates responses.
* **Destruction**: Servlet's destroy() method is called when servlet is removed, or container shuts down.

**Que: Difference between @RequestParam and @PathVariable annotations in Spring MVC.**

**Ans:**

**@RequestParam:**

* Used to extract parameters from the query string or form data of an HTTP request.
* Parameters are specified as key-value pairs in the URL or form submission.
* Usage: Suitable for optional or query parameters passed in the URL, such as ?id=123.
* Syntax: @RequestParam("paramName") String paramName

**@PathVariable:**

* Used to extract parameters from the URL path itself.
* Parameters are embedded in the URL path as part of the resource identifier.
* Usage: Ideal for mandatory or path-based parameters included directly in the URL path, such as /products/123.
* Syntax: @PathVariable("paramName") String paramName

In summary, @RequestParam is used for query parameters passed in the URL or form data, while @PathVariable is used for path-based parameters embedded in the URL path itself.

**Junit on Mockito**

**Que: What is Junit?**

**Ans:** JUnit is a popular open-source testing framework for Java that is widely used by developers for writing and running automated unit tests.

JUnit is specifically designed for unit testing, where individual units of code (such as methods or classes) are tested in isolation to verify their functionality.

JUnit helps to increase the software stability as it helps in identifying the bug in the code logic at an early stage without requiring the software to go to production.

**Que: What is Unit Testing?**

**Ans:** Unit testing is a software testing strategy that tests single entities like methods or classes at a time. This helps to ensure that the product quality is met as per the business requirements.

**Que: Why do we use JUnit? Who uses JUnit more - Developers or Testers?**

**Ans**: JUnit is used more often by developers for implementing unit test cases for the functionalities they have developed.

JUnit is used due to the following reasons:

* Helps in automating test cases.
* Helps in reducing defects in the code logic whenever the logic changes.
* Helps in reducing the cost of testing as the bugs are identified, captured and addressed at early phases in the software development.
* Helps to identify the gaps in coding and gives a chance to refactor the code.

**Que: What are the features of JUnit?**

**Ans:** Following are the features of JUnit:

JUnit is an open-source framework.

Supports automated testing of test suites.

Provides annotations for identifying the test methods.

Provides assertions to test expected results or exceptions of the methods under test.

Provides a platform for running test cases automatically and checking their results and giving feedback.

**Que: Is it mandatory to write test cases for every logic?**

**Ans:** No, it is not mandatory. However, test cases can be written for the logic which can be reasonably broken and tested independently.

**Que: How will you Implement Junit in Spring Boot?**

**Ans:** To implement JUnit testing in a Spring Boot application, you can follow these steps:

**1. Add JUnit and Spring Boot Test Dependencies:**

***<dependency>***

***<groupId>org.junit.jupiter</groupId>***

***<artifactId>junit-jupiter-api</artifactId>***

***<version>5.8.1</version>***

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

***</dependency>***

***<dependency>***

***<groupId>org.junit.jupiter</groupId>***

***<artifactId>junit-jupiter-engine</artifactId>***

***<version>5.8.1</version>***

***<scope>test<***

**2. Write JUnit Test Cases:**

* Create test classes for your Spring Boot application components, such as controllers, services, repositories, etc. Annotate your test classes and methods using JUnit annotations.

**3. Run JUnit Tests:**

* Use your IDE's testing support or Maven/Gradle commands to run the JUnit tests.
* In IntelliJ IDEA or Eclipse, you can right-click on the test class or method and select "Run" or "Debug" to execute the tests.
* Using Maven, run mvn test in the terminal to execute the tests.

**4. Review Test Results:**

After running the tests, review the test results to ensure that your Spring Boot components are functioning as expected. JUnit will report pass/fail results based on the assertions in your test cases.

**Que: How to write test Case?**

**Ans:**

1. First we create a Test Class for the class which we want to test

2. Create an object of that class.

3. @BeforeEach annotated method runs before each test case.

4. @Test method to indicate that the method uses test method.

5. @DisplayName is used for defining the test name displayed to the user.

6. assertEquals() method is used for validating whether the expected and actual values are equal.

7. @RepeatedTest annotation indicates that the test method will be run 5 times.

**Que: What are some of the important annotations provided by JUnit?**

**Ans**: Some of the annotations provided by JUnit are as follows:

**@Test:** This annotation over a public method of void return type can be run as a test case.

btayega ki ye test method hai.

**@Before or BeforeEach**: This is used when we want to execute the preconditions or any initialisation based statements before running every test case.

Example: Initializing objects, setting up test data, resetting state, etc.

Ye test case se pahle run hoga yahi method yahi uss method ko run krega aur usko object banayega jisko hume test krna hi.

**@BeforeClass:** This is used when we want to execute statements before all test cases. The statements may include test connections, common setup initialisation etc.

Example: Initializing shared resources, setting up static variables, starting external services, etc.

static variable wagarh set kren ke liye hota hai.. eek hi bar chalta hai pure testing me ... jaise static variable eek hi bar banta haiw wahi bar bar use hota hai.

**@After:** This is used when we want to execute statements after each test case. The statements can be resetting the variables, deleting extra memory used etc.

ye har test case ke run hone ke baad chaleag taki hm value change kr sake dusre test ke liye.

**@AfterClass**: This is used when certain statements are required to be executed after all the test cases of the class are run. Releasing resource connections post-execution of test cases is one such example.

**@Ignores**: This is used when some statements are required to be ignored during the execution of test cases.

**@Test(timeout=x):** This is used when some timeout during the execution of test cases is to be set. The value of x is an integer that represents the time within which the tests have to be completed.

ye test ka time set krne ke liye hai isse zzada time lega toh test case fail ho jayega.

**@Test(expected=NullPointerException.class):** This is used when some exception thrown by the target method needs to be asserted.

**@Test(expected):**

Specifies an exception type that a test method is expected to throw.

Used to verify that a test method throws a specific exception, and the test will fail if the exception is not thrown.

**@RunWith** annotation lets us run the JUnit tests with other custom JUnit Runners like SpringJUnit4ClassRunner, MockitoJUnitRunner etc. We can also do parameterized testing in JUnit by making use of @RunWith(Parameterized.class).

Simple example:

public class CalculatorTest {

Calculator calcObject;

@BeforeEach

void setUp() {

calcObject = new Calculator();

}

@Test

@DisplayName("Add 2 numbers")

void addTest() {

assertEquals(15, calcObject.add(10, 5));

}

@RepeatedTest(5)

@DisplayName("Adding a number with zero to return the same number")

void testAddWithZero() {

assertEquals(15, calcObject.add(0, 15));

}

}

**Que: What is mocking and stubbing?**

**Ans:** Mocking and stubbing are techniques commonly used in software testing, especially in unit testing, to simulate behavior and control dependencies. Both techniques are used to create test doubles or mocks of objects that are used by the code under test, allowing for isolated and controlled testing environments.

**Mocking:**

Mocking is the process of creating fake objects or mocks that mimic the behavior of real objects but are designed specifically for testing purposes.

Mock objects are used to simulate interactions with external systems, collaborators, or dependencies, such as databases, web services, or external APIs.

Mocking frameworks (e.g., Mockito, EasyMock) provide utilities for creating mock objects, defining behavior, verifying interactions, and setting expectations.

fake object banate hai..

Stubbing:

Stubbing is a specific aspect of mocking, where you define the behavior and responses of mock objects when certain methods are called or certain conditions are met.

Stubbing involves setting up expectations and return values for method calls on mock objects, allowing you to control the behavior of the mock during testing.

stubing moking ka hi part hai jisme hum mock object ke return value and behavior ko define krte hai..

import static org.mockito.Mockito.\*;

// Create a mock object

MyService myServiceMock = mock(MyService.class);

// Define mock behavior (stubbing)

when(myServiceMock.getValue()).thenReturn(10);

// Verify interactions (mock verification)

verify(myServiceMock).getValue();

To achieve this we nee to add Mockito Dependency:

First, you need to add the Mockito dependency to your project. For Maven projects, include the following dependency in your pom.xml file:

xml

Copy code

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>3.12.4</version> <!-- Replace with the latest version -->

<scope>test</scope>

</dependency>

**Que: How to ignore tests in JUnit?**

**Ans:** We need to ignore test cases when we have certain functionalities that are not certain or they are under development. To wait for the completion of the code, we can avoid running these test cases. In JUnit 4, we can achieve this by using @Ignore annotation over the test methods. In JUnit 5, we can do it using @Disabled annotation over the test methods.

**Que: Define code coverage. What are the different types of code coverages?**

**Ans:** Code coverage is a metric used in software testing to measure the extent to which the source code of a program has been executed during testing. It helps assess the effectiveness of test cases by identifying areas of code that have not been executed, indicating potential gaps in the testing process. Code coverage is typically expressed as a percentage, representing the ratio of executed code to the total codebase.

Statement coverage measures the percentage of executable statements in the code that have been executed during testing.

**1. JaCoCo (Java Code Coverage):**

JaCoCo is a widely used code coverage tool for Java applications, including Spring Boot projects.

It provides detailed coverage reports in various formats, such as HTML, XML, and CSV, allowing developers to analyze code coverage metrics.

**2. EclEmma (Eclipse Code Coverage):**

EclEmma is a code coverage tool specifically designed for Eclipse IDE users.

It integrates seamlessly with Eclipse, allowing developers to run tests and generate coverage reports directly within the IDE.

**Install EclEmma Plugin:**

* Open Eclipse IDE and go to Help -> Eclipse Marketplace.
* Search for "EclEmma" in the Marketplace search bar.
* Install the "EclEmma Java Code Coverage" plugin.
* Follow the on-screen instructions to complete the installation and restart Eclipse if prompted.

**Configure EclEmma for Your Project:**

* Open your Java project in Eclipse.
* Right-click on the project or the package containing your source code.
* Go to Properties -> Java Build Path -> Source tab.
* Ensure that your source folders are included in the build path.

**Run Your Tests with Coverage:**

* Write JUnit test cases for your project's classes and methods, or ensure that you have existing test cases.
* Right-click on your test class or package containing test classes.
* Select Run As -> JUnit Test to run your tests.

**Que: Why do we need mocking in unit testing?**

**Ans**: Mocking is the process of creating and using mock objects that simulates the behaviour of real objects and is used to isolate the behaviour of the module under test from its external dependencies or services. These are particularly useful in unit testing and help in making unit test cases repeatable and predictable.

Unit testing focuses on testing individual units (e.g., classes, methods) of code in isolation.

Mocking allows developers to isolate the unit under test by replacing real dependencies (e.g., external services, databases, APIs) with mock objects.

This isolation ensures that failures or changes in external dependencies do not impact the unit test results, making tests more reliable and consistent.

**Que: What is Mockito?**

**Ans:** Mockito is a popular open-source Java library used for creating mock objects and performing mocking in unit tests.

Mockito provides utilities for creating mock objects of interfaces, abstract classes, and concrete classes.

Mock objects mimic the behavior of real objects but are specifically designed for testing purposes, allowing developers to isolate and control dependencies.

**Que: What are some annotation used for Mockito?**

**Ans:** Some of the commonly used annotations in Mockito are:

**1. @Mock:**

Purpose: Used to create mock objects.

@Mock

private MyService myServiceMock;

isse object banta hai

**2. @Spy:**

Purpose: Used to create spy objects (partial mocks) that wrap real objects and allow selective mocking of methods.

@Spy

private MyService myServiceSpy;

isse kch custon object banta hai

**3. @InjectMocks:**

Purpose: Used to inject mocks into the target test class (the class being tested).

isse hum mock object ko test class me inject krte hai..

**4. @Captor:**

Purpose: Used to capture method arguments for verification in test cases.

result capturree krne ke liye

**5. @MockBean (Spring Boot):**

Purpose: Used in Spring Boot tests to create mock beans for autowired dependencies.

test class me define krte hai.. for autowiring ke liye.

@MockBean (Spring Boot):

Annotation: @MockBean

Purpose: Used in Spring Boot tests to create mock beans for autowired dependencies.