Spring Boot Interview Questions

Q 1. Why will you choose Spring Boot over Spring Framework?

- There are several reasons to choose Spring boot over the traditional Spring framework.

1. Faster Development: Spring boot simplifies the setup and configuration of Spring applications by providing default configurations, reducing boilerplate code. With Spring, developers have to manually configure XML or java-based configurations, but Spring Boot offers automatic configuration.
2. ~~Embedded Server~~ Microservice-friendly: Spring boot is tailored for microservices architecture. It provides inbuilt tools to create RESTful APIs, and it's easy to manage dependencies for building microservices compared to the traditional Spring framework.
3. Embedded Server: Spring boot comes with embedded servers like Tomcat, Jetty, so there's no need to deploy the application to the external server. This speeds up the development cycle as you can run the application directly using the java -jar command.
4. Production-Ready features: Spring boot includes several production-ready features out of the box, such as health checks, metrics, and monitoring through Actuator.
5. Dependency Management: Spring boot's dependency management simplifies version control through its starter dependencies. It manages versions of common libraries, ensuring compatibility without the developer manually managing individual dependencies.

6- Opinionated Defaults : Spring boot follows "convention over configuration", meaning it provides opinionated ~~without the developer manually~~ defaults that work in most scenariors. If necessary, developers can still override these configurations, making it both flexible and easy to use.

Q 2- What all spring boot starter you have used or what all modules you have worked on?

- These are ~~for~~ spring boot starters

1- Spring boot starter web

- To build RESTful APIs and web applications.
- Provides necessary libraries like Spring MVC and embedded Tomcat sever to create web applications quickly.
- Creating both synchronous and assynchronous API endpoints, handled request/response mapping, and worked with various HTTP ~~meid~~ methods like GET, POST, PUT and DELETE.

2- Spring boot starter JPA-

This starter is used for interacting with databases using the JPA (Java Persistence API). I've worked with Hibernate as the JPA provider to manage database entities, implement repositories and handle CRUD operations. Also utilized @Query annotation for custom queries and worked with various JPA features like pagination, lazy loading and transactions.

3- Spring boot Starter AOP (Aspect-Oriented Programming)
- Used AOP for cross-cutting concerns such as logging, security and performance monitoring.
- For example, I used AOP to create custom annotations for logging method creation times and auditing. This helped me separate concerns and avoid repetitive code throughout the application.

4- Spring boot Starter Web Services
- Used for SOAP (Simple Object Access Protocol) based web services.
- Built & utilized SOAP services using @SoapAction and @Endpoint.

5- Spring boot starter Security
- Used for implementing security in applications.
- Authentication and Authorization using Spring Security.
- Worked with role-based access control, JWT token and OAuth2.

6- Spring boot Starter thymeleaf
- Used Thymeleaf as the templating engine for server-side rendered web applications.
- Helps developing server side web pages.

Q3- How will you run a Spring Boot Application

1- Running from the IDE - If working on Intellij or Eclipse then I can run the main class annoated with @SpringBootApplication. It will start the embedded tomcat sower. Useful for development environment.

2- Using Maven or Gradle from Command line:
- For maven mvn spring-boot:run builds & runs the project in one step.
- For gradle ./gradlew bootRun.

3- Running JAR file:
- Package it as executable JAR by running mvn clean package. or ./gradlew build.
- Once jar is built run java -jar target/myapp.jar.

4- Deyloying WAR file:
- package the app as WAR file and run it to an external tomcat or Jetty Server.

Why prefer JAR Over WAR ?
- Includes embedded Server. No need to saperately manage tomcat or any othe containers.
- However, in environment where external server is already setup and organization only wants WAR then WAR is used

Q4- Purpose of using @SpringBootApplication annotation

- The @SpringBootApplication annotation is essentially a convenience annotation in Spring Boot that combines three key annotations into one. It's primary purpose is to bootstrap and auto-configure a Spring Boot application. It simplifies configuration by eliminating the need to manually define the same settings repeatedly.

Breakdown of 3 annotations it encapsulates:

1- @EnableAutoConfiguration: This annotation tells Spring boot to automatically configure the application based on the dependencies on the classpath. For example, if I have spring-boot-starter web in my project, it will configure an embedded Tomcat server, MVC controllers and other web-related components without needing manual configuration.

2- @ComponentScan: This enables Spring to Scan the package (and sub-packages) where the @SpringBootApplication class resides, looking for components such as @Controller, @Service, @Repository. It automatically detects beans and registers them in the Spring Context.

3- @Configuration: This allows the class to be used as source of bean definitions. It's equivalent of manually creating a configuration class and marking it with @Configuration.

- Using @SpringBootApplication, I can start application and it'll take care of component Scanning, auto-configuration, and bean management.

Q4- Can I use 3 annotations instead of @SpringBootApplication

- Yes, you can
- Cases where you might need to do that -

1. you might want to customize the component scan path or control exactly which parts of auto-configurations are enabled.
2. Want to configure each annotation seperately.

Q5- Auto-configuration in Spring Boot.

- Automatically configures your Spring application based on the dependencies present on the classpath. It aims to simplify development by reducing manual configuration.

How it works -

- Classpath Scanning - Spring boot examines the libraries and dependencies that are present on the classpath. Based on what it finds, it auto-configures beans and other settings.
- For example, if you include Spring-boot-starter-web, Spring boot automatically configures an embedded Tomcat server, a DispatcherServlet, and sets up Spring MVC for you.
  If you include Spring-boot-starter-data-jpa, it will auto-configure an EntityManagerFactory, JPA repositories, and other database-related beans.

- Conditional Configuration: Auto-configuration is smart enough to not override any existing configurations. It uses a series of @Conditional annotations to ensure that auto-configuration only happens when necessary

and no existing beans or configurations are defined by the developer.

- For example, if you've already configured a DataSource bean, Spring boot will not create one automatically.

- META-INF/spring.factories: Behind the scenes, auto-configuration is managed by classes listed in the spring.factories file inside the JAR.

- Customization: Auto-configuration simplifies things but its also flexible. You can override any auto-configuration by defining your own bean. You can also exclude auto-configuration classes if you want to disable certain behavior using
@EnableAutoConfiguration(exclude = {SomeAutoConfiguration.class})

Example -
When working with databases, if you include Spring-boot-starter-data-jpa and have database driver like MYSQL on the classpath, Spring will automatically configure a Datasource, an EntityManager factory and establish a connection pool for you.

Q6- How to disable a specific auto configuration class.
2 ways:
1- Using @SpringBootApplication or @EnableAutoConfiguration
use → @EnableAutoConfiguration(exclude = {HibernateJPAAutoConfiguration.class})

2- Using application.properties or application.yml

Spring:
autoconfigure:
exclude:
- org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaConfiguration

Q7- How to customize the default configuration in Spring Boot?
- You can customize beans, properties, or even replace the default behaviour with your own implementations.

1- Using application.properties or application.yml
- Customize the default server port
server.port=8081
- Configuring a custom datasource
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=root
spring.datasource.password=pass123
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

2- Custom Bean Definition
- to bypass auto-configuration we can use our custom beans. If same bean already exists in auto-config then it'll be overridden by your custom definition.

Example of RestTemplate

```
@Configuration
public class MyConfig {
  @Bean
    public RestTemplate restTemplate() {
      RestTemplate restTemplate = new RestTemplate();
        // custom code.
      return restTemplate;
} }
```

3- Using @ConfigurationProperties
- Spring boot allows you to map properties directly to POJOs using @ConfigurationProperties.
- Example of custom mail configuration

```
mail.host=smtp.gmail.com
mail.port=587
mail.username=root
mail.password=root
```

```
Property class
@Component
@ConfigurationProperties(prefix="mail")
public class MailProperties {
    private String host;
    private int port;
    private String username;
    private String password;
}
```

4- Using @Contitional Annotation
When you want to customize behaviour for a specific condition.

```
@Bean
@Conditional(MyCustomcondition.class) // create this bean only in this condition
public MyService myService(){
    return new MyService();
}
```

5- Custom Auto-Configuration Classes
- Create your own @Configuration class and replace the beans that are part of auto-configuration.

Example of Custom JPA Hibernate Configuration

```
@Configuration
public class MyJpaConfig {

    @Bean
public JpaTransactionManager transactionManager(EntityManagerFactory emf)
{
    JpaTransactionManager txManager = new JpaTransactionManager();
    txManager.setEntityManagerFactory(emf);
    //custom code----
    return txManager;
}
}
```

Q 8- How run() method in Spring boot works internally?
1- Creates a SpringApplication instance — initilizes and configures the application.
2- Set up the environment — loads properties, configures profiles and prepares the environment.
3- Creates the ApplicationContext — Depending on the application type (web, non-web, reactive), Spring Boot creates the appropriate ApplicationContext.
4- Registers listeners and initializers - Handles events during the application lifecycle.

5- Refreshes the ApplicationContext - initializes beans, resolve dependencies, setup the entire Spring container.
6- Starts the embedded web server - if it's web application, the embedded server (eg Tomcat) starts.
7- Runs CommandLineRunners and ApplicationRunners - Executes any application-specific startup logic.
8- Application runs - The application is up and running, and a shutdown hook is registered for graceful shutdown.

Q9. What is CommandLineRunner in Spring Boot?
- It is a functional interface provided by Spring boot, which is used to execute code after the Spring Boot application has started. It's often used for application initilization tasks such as loading data into the database or setting up certain configurations.
- To use CommandLineRunner, you implement Run() method, where you can write logic that needs to be executed at startup.

Example

```
@Component
public class MyStartUPRunner implements CommandLineRunner{
  @Override
  public void run(String... args) throws Exception{
    Sout("App started with arguments :"+ Assays.toString(args));
        // custom logic
  }
}
```

Q10- Purpose of Stereotype annotations in the Spring Framework?

- Stereotype annotations in Spring are used to define the roles or responsibilities of classes in the application content. These annotations serves a way to inform the Spring container about the purpose or the function of a class, and spring will automatically detect and manage them during component scanning.

The most common stereotype annotations are:

1- @Component: This is the most generic stereotype annotation or parent of all stereotype annotation, which indicates that a class is a spring-managed component. It tells Spring to treate this class as a bean and register it in the application context.

2- @Service: This is a specialized form of @Component, used specifically to indicate that a class performs business logic or service tasks. Indicates that this class is a part of service layer.

3- @Repository: This annotation is used to define a class as a Data Access Object (Dao), which interacts with the database. Additionally, it provides exceptions translation, converting persistence exceptions into Spring's data access exceptions.

4- @Controller: This annotation indicates that a class serves as a controller in the Spring MVC framework, handling HTTP requests, typically returning views.

Difference between @Component & @Bean

- @Component is a class-level annotation used to mark a class as a Spring-managed bean. It is used to automatically detect and register beans with Springs component scanning mechanism, Spring will create an instance of the class and manage it in the application context.
- @Bean is a method-level annotation used to explicitly declare a bean in a spring configuration class. Typically used inside class annotated with @Configuration. It returns instance of the bean and Spring registers it in the application context. Use it for custom logic such as database connection pool or library that spring boot automatically scans for.

Q11- How to define a Bean?

Two important ways -

1- Using @Component and Stereotype annotation
2- Using @ Configuration & @Bean (Java-based).

Service class -

```
public class DemoService{
}
```

Config class or Bean class -

```
@Configuration
public class AppConfing{
    @Bean
    public DemoService demoService(){
        return new DemoService();
    }
}
```

Q12- What is Dependency Injection?

- DI is a design pattern used in Spring Boot to achieve loose coupling between objects. It allows an object (the dependant) to have its required dependencies provided by an external source, rather than creating them within the object itself. It basically promotes loose coupling.

Types:

1- Constructor injection: Dependencies provided via a constructor.

Example:

```
@Service
public class MyService {
    private final MyRepository repository;

    @Autowired
    public MyService(MyRepository repository) {
        this.repository = repository;
    }
```

2- Setter injection: Dependencies are injected via setter methods.

Example:

```
@Service
public class MyService {
    private MyRepository repository;
    @Autowired
    public void setRepository(MyRepository repository) {
        this.repository = repository;
    }
}
```

3- Field injection: Dependencies are injected directly into fields.

Example:

```
@Service
public class MyService{

    @Autowired
    private MyRepository repository;
}
```

In Summary, DI means using a dependency object without taking care of the creation of that object. @Autowired is used for that, it tells Spring to manage the Bean.

Definition of a Bean

- In Spring, a Bean is an object that is managed by the Spring Inversion of Control (IoC) container.

Essentially, it is a fundamental building block in a Spring application. A bean can be any Java object, but Spring is responsible for the lifecycle, configuration and dependencies of these objects.

Q13. Where to use setter injection over constructor injection, and vice versa?

- Constructor injection is better for mandatory dependencies and promotes immutability.
- Setter injection is useful for optional dependencies or cases where you need flexibility to reassign dependencies later.
- Field injection is the quickest one but generally not recommended because of its impact on testability, immutability and clarity of code. They are mutable, hard to test, might create circular dependency and doesn't force initilization of dependencies.

Q14 @PostConstruct real-world use case?

- @PostConstruct is an annotation used to define pre processing logic or to execute some code at application startup.
- It works similar to CommandLineRunner interface, we just use it to make the use of annotation instead of using any interface.

Example

```
@SpringBootApplication
public class MainClass {
    @PostConstruct
    public void printSomething() {
        Sout("Something");
    }

    psvm ----
}
```

Q15- How to dynamically load values in Spring boot app?

- Using @value And load from application.properties

```
@Value ("${app.name}")
  String name;
```

Using @ConfigurationProperties

```
@ConfigurationProperties(prefix = "app.config")
```

Q16- Key difference between YAML and properties file, when to prefer one over the other

- YAML is better for configuration where hierachical clarity and structured data are essential. It is preffered when you need better readability for complex configurations involving lists, nested objects or multiple values.
- properties is better when you want simplicity and don't want to worry about formatting errors due to whitespace. Use it when your configuration is simple, flat or when backward compatability or legacy constraints dictate it.

YAML Example-

```
app:
  name : MyApp
  version : 1.0
  database :
    url: jdbc:mysql://localhost:3306/mydb
    username : root
    password : pass
```

application.properties Example:

```
app.name = MyApp
app.version = 1.0
app.database.url = jdbc:mysql://localhost:3306/Mydb
app.username = root
app.password = pass
```

Q17. Difference between YML and YAML?

No difference

yml is a shorter alternative and it was used earlier due to system restrictions of having three letters in extension name. Now there's is no such restrictions so both can be used and it'll not make any difference.

Q18- If you configure same values in .yml and .properties file then which one will be picked by Spring boot?

- Spring boot by default it loads .properties first and .yml later so .properties value will get override by .yml file and .yml config will be used as it loads later.
- If both files contains same properties, the value from application.yml will be used, as it loaded later in the order of precedence.

Q19- How to load external properties in Spring boot?

- Suppose if you don't want to define anything in .properties or .yml file and you want to read properties from a file that is external to the project then to achieve this you need to create one external application.properties file then inside your main application.properties file →

```
spring.config.import=file://location-in-your-system
```

Q20- How to map or bind config properties to java object?

- use @ConfigurationProperties(prefix = "example.prefix")
- Example is given earlier in question 7.

Q21- How will you resolve bean dependency ambiguity?

- Bean dependency ambiguity occurs when ~~same type bean~~ there are multiple beans of the same type and the framework is unsure which bean to inject.
- It can be resolved by using @Qualifier annotation.

Example:

```
@Bean
public Service service1(){
    return new ServiceImpl1();
}

@Bean
public Service service2(){
    return new ServiceImpl2();
}
```

```
@Autowired
@Qualifier("service1")
private Service myService;
```

Q22. Can we avoid this dependency ambiguity without using @Qualifier?

- Yes, by using @Resource from jakarta.annotation

```
@Autowired
@Resource(nam="service1")
private Service myService;
```

@Qualifier is specific to the Spring boot but @Resource is given by java itself.

Q23: What is bean scope and explain different types of bean Scope?

- Bean scope refers to the lifecycle of a bean, or how long it lives and how it is shared within the application. The scope determines when a bean is created, how long it exists and how it behaves within the IOC container. (Inversion of Control).

Types:

1- Singleton Scope: it is default scope, if you don't define then automatically bean is singleton scope.
Only one instance created and shared throughout the application.
It is created when spring application content is initialized and lives for the application duration.

2- Prototype : A new instance is created everytime it is requested from the Spring container. Spring container creates this bean and not responsible for its lifecycle.

Example:

```
@Bean
@Scope("prototype")
public Service service(){
    return new ServiceImpl();
}
```

3- Request Scope: One instance created for each HTTP request. Once request is completed, the bean is discarded. Bean is created at start of request and end at end of request.

Example:

```
@Bean
@Scope(value = WebApplicationContext.SCOPE_REQUEST, proxyMode =
                                        ScopedProxyMode.TARGET_CLASS)
public MyService myRequestScopedService(){
    return new MyServiceImpl();
}
```

4- Session Scope: Created for HTTP session and gets destroyed after that. Useful for user session data.

Example:

```
@Bean
@Scope(value = WebApplicationContext.SCOPE_SESSION, proxyMode = ScopedProxyMode.TARGET_
                                                                CLASS)
public MyService mySessionScopedService(){
    return new MyServiceImpl();
}
```

5- Application Scope: Similar to Singleton scope but specific to web applications. Created when ServletContext is initialized and destroyed when ServletContext is terminated.

Example:

```
@Bean
@Scope(value = WebApplicationContext.SCOPE_APPLICATION)
public MyService myAPPService() {
    return new MyServiceIMPL();
}
```

Q24- Define custom Bean Scope?

1- Implement the Scope interface: Create a class that implements org.springframework.beans.factory.config.Scope interface. This define the lifecycle methods for beans.

```
public class CustomScope implements Scope {
 private Map<String, Object> beans = new HashMap<>();
 @Override
public Object get(String name, ObjectFactory<?> objectfactory) {
   return beans.computeIfAbsent(name, k -> objectfactory.getObject());
 }
 @Override
 public Object remove(String name){
     return beans.remove(name);
  }
}
```

2- Register the Custom Scope: Register your custom scope in the Spring Context using a CustomScopeFigurer.

```
@Bean
public CustomScopeFigurer customScopeFigurer() {
    CustomScopeFigurer configurer = new CustomScopeConfigurer();
    Configurer.addScope("custom", new CustomScope());
    return configurer;
}
```

3- Use it:

```
@Bean
@Scope("custom")
public MyService myService() {

    return new MyServiceImpl();
}
```

Q:25- Real-Time use case of Singleton Scope and Prototype scope?

Singleton	Prototype
1- Database Configuration	User Services
2- Service layer	Thread safety
3- Application Configuration	Heavy initialization

Q26- Can we inject prototype bean in Singleton bean? If yes, What will happen if we do that?

- If you inject prototype bean in singleton bean then it will lose it's scope and behave like Singleton bean.

Example:

```
@Component
@Scope("prototype")
public class PrototypeBean{
  public PrototypeBean(){
     Sout("Prototypebean Instantiate");
  }
}

@Component
public class SingletonBean{
   @Autowired
   private PrototypeBean prototypeBean;

   public SingletonBean(){
     System.out.println("SingletonBean() instantiated !!");
   }

   public PrototypeBean getPrototypeBean(){
      return prototypeBean;
   }
}
```

Q27- Difference between Spring Singleton and plain Singleton?

- Spring singleton means one instance per Spring Container. If you create multipe container then youll get multiple instances.
- Plain singleton means One instance per JVM (Classloader).

Q28- Purpose of BeanPostProcessor interface in Spring, how to use it to customize bean initilization and destruction?

BeanPostProcessor provides hooks and callbacks to perform pre-processing logic. It allow us to perform custom actions on bean initilization. (before & after initilization as well)

```
@Component
public class MyBeanPostBrocesser implements BeanPostProcessor {
@Override
public Object postProcessBeforeInitilization(Object bean, String beanName){
  System.out.println("Before Init" + beanName);
    return bean;
}
@Override
public Object postBrocessAfterInitilization(Object bean, String beanName){
  System.out.println("After init" + beanName);
    return bean;
}
}
```

Q29- Have you worked on RESTful web Services? If yes then list methods.

- Yes, for example in user management system:
  - GET /users to retrieve all users.
  - POST /users to create a new user.
  - PUT /users/{id} to upadat a specific user
  - PATCH /users/{id} to partially update a specific user.
  - DELETE /users/{id} to delete a specific user.

Q30- How to specify HTTP method type for a REST endpoint?

- User management example

```
@RestController
@RequestMapping("/users")
public class UserController{
  @GetMapping
  public List<User> getAllUsers() {
      // Logic to retrive all users
  }

  @PostMapping
  public User createUser(@RequestBody User user){
      // Logic to create a new user
  }

  @PutMapping("/{id}")
  public User updateUser(@PathVariable Long id,@RequestBody User user)
{
    // Logic to update a user
}
```

```
@PatchMapping("/{id}")
public User partialUpdateUser(@PathVariable Long id, @RequestBody Map<String,Object>
{
    // Logic to partially update a user
}

@DeleteMapping("/{id}")
  public void deleteUser(@PathVariable Long id){
      // Logic to delete a user
  }
}
```

Q 31- Design a rest endpoint, Assume you have a Product database, and your task is to create an API to filter a list of products by productType.

— Product Entity Class:

```
@Entity
public class Product{
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  private String name;
  private String productType;
  private Double price;

    // Getters and Setters
}
```

Product Repository:
Repository interface to interact with database

```
public interface ProductRepository extends JPARepository<Product, Long>{
  List <Product> findByProductType(String productType);
 }
```

Service Class:
Service class to handle business logic:

```
@Service
public class ProductService{
  @Autowired
 private ProductRepository productRepository;

public List<Product> getProductsbyType(String productType){
  return productRepository.findByProductType(productType);
   }
 }
```

Controller:
Controller define the REST Endpoint.

```
@RestController
@RequestMapping("api/products")
public class ProductController {

@Autowired
private ProductService productService;
```

```
@GetMapping
public List<Product> getProductsByType(@RequestParam String input){
  return productService.getProductByType(productType);
}
}
```

Q. 32- Customize the above code to provide all products if user does not provide productType.

- Change in Service

```
public List<Product> getProductsByType(String productType){
 if(productType != null && !productType.isEmpty()){
   return productRepository.findByProductType(productType);
 }
 return productRepository.findAll();
}
}
```

- Change in Controller -

```
@GetMapping
public List<Product> getProducts(@RequestParam(required=false) String input){
  return productServic.getProductsByType(input);
}
}
```

Q33 Difference between @PathVariable & @RequestParam?

- @PathVariable extracts value from the uri path.
  /users/{id} → users/10 where id = 10.
- @RequestParam extracts value from the query parameters.
  /users ?id=10 where id = 10.

In summary, @PathVariable is for path segments, and @RequestParam is for query parameters.

Q34 Why @RestController over @Controller

- @RestController is a specialized version of @Controller that combines @Controller and @ResponseBody. It simplifies the creation of RESTful APIs by automatically converting the return value of methods into JSON or XML responses without the need for explicit @ResponseBody annotation.
- @Controller typically used for MVC applications where you return views (HTML pages) rather than data.
- @RestController returns data in JSON or XML & used for REST APIs

- If you use @Controller then you need to use @ResponseBody annotation on every controller method.

Q35- How can we deserialize a JSON request payload into an object within a Spring MVC controller?

- Using @RequestBody annotation. It tells Spring to convert the incoming JSON into the specified Java object.

Q36. Can we perform update operation in POST http method if yes then why do we need PUT mapping or put http method.

- Yes you can, but it's not recommended as per RESTful principles.
post is not idempotent, means multiple identical request can result in different outcomes.
put is idempotent, means if you provide same request again and again then it'll have same affect and nothing will change.

Q37. Can we pass Request Body in GET HTTP method?

- NO, according to HTTP/1.1 specifications, the GET method is not designed to carry a request body. GET requests are intented to retrieve data and should pass any necessary information via the URL (Path or query parms).

Q.38 String interning in Java?

- Storing the heap area stored String into the Stringpool is String interning.

```
String s1 = new ("Hello");  //created in heap

String s2 = s1.intern();  // created in String pool
```

Q39. How to pass variable number of arguments in method?

- public static void sum(int... numbers)
this is called varargs

Q40- Contentent negotiation (XML/JSON) in REST endpoint?

- Content negotiation allows REST endpoint to return data in different formats (JSON or XML)

@GetMapping(produces = {"application/json", "application/xml"})

jackson package in java core library automatically handles the JSON but to allow use of XML you need to use jackson-dataformat-xml package in your pom.xml file.

Q41- What all status codes you've observed in your projects?

1- 2XX Success codes

- 200 OK: The request was success and response contains requested data.
- 201 Created: Successful & new resource was created - Used in post request mostly.
- 304 No content: Successful but no content to return. Often used in delete requests.

2- 3xx Redirection codes

- 301 moved permanently: This resource permanently moved to new url. Client should update it's url.
- 302 Found: The resourse is temporarily located at a different Url. Client should follow the redirect but continue to use original URL for future requests.

3- 4XX Client Error codes

- 400 Bad request: Server could not understand the

request due to invalid syntax.

- 401 unauthorized: The request requires user authentication, but the client did not provide valid credentials.
- 403 Forbidden: Server understand the request but refuses to authorize it. Client do not have permission to access the resource.
- 404 Not Found: requested resource could not be found on the server.
- 409 conflict: The request could not be completed due to a conflict with the current state of the resource. (i.e., trying to create a duplicate resource).
- 405 method not allowed: when you provide wrong request type.

4- 5xx Server Error Codes: ~~The server encountered an~~

- 500 Internal Server error: The server encountered an unexpected condition that prevents it from fulfilling the request.
- 502 Bad Gateway: The server, while acting as a gateway received an invalid response from the upstream server.
- 503 Service unavailable: This server currently unable to handle the request due to temporary overload or maintainence.

Q42 How to customize status code for your endpoints?

1- Using @ResponseStatus Annotation

```
@GetMapping("product/{id}")
@ResponseStatus(HttpStatus.OK) // custom status code for this method
public Product getProduct(@PathVariable Long id){
    return productService.findById(id);
}
}
```

2- Returning a ResponseEntity
ResponseEntity<T> allows to specify bothe the response body and Status code.

```
@PostMapping("/products")
public ResponseEntity<Products> createProduct(@RequestBody Product product){
Product createProduct = productService.save(product);
  return new ResponseEntity<>(createProduct, HttpStatus.CREATED);
}
}
```

3- Using Exception Handling with @ExceptionHandler

Q47 How to enable Cross Origin?
- To enable Cross-Origin Resource Sharing:
1- Using @CrossOrigin Annotation

```
@RestController
@CrossOrigin(origin = "http://example.com")
public class MyController{

}
```

2- Global CORS Configurations using @Configuration class

3- Using Spring boot Properties in application.properties
spring.web.cors.allowed-origin-patterns = http://example.com

Q44 How to upload file in Spring Boot?

- Multipart is used REST controller to upload files in Spring boot.

Steps for file upload:

1. Add dependency:
   spring-boot-starter-web

2. Configure file upload in application.properties
   spring.servlet.multipart.enabled = true
   spring.servlet.multipart.max-file-size = 2MB
   spring.servlet.multipart.max-request-size = 2MB

3. REST Controller to handle file upload
   use the MultipartFile interface in the controller

```
@RestController
@RequestMapping("/files")
public class FileUploadController{
  @PostMapping("/upload")
public ResponseEntity<String> uploadFile(@RequestParam("file") MultipartFile fi
 if(!file.isEmpty()){
    try{
    byte[] bytes = file.getBytes();
   return ResponseEntity.OK("File uploaded", + file.getOriginalFileName());
}catch(Exception e){
   return ResponseEntity.status(500).body("failed to upload");
}
return ResponseEntity.badRequest().body("File is empty");
}
```

Q45. How to maintain versioning for your REST API?

- Versioning is used when you have major changes in an endpoint and you don't want to remove the previous code then you can define the version for that endpoint

Approaches to take for versioning of API

1- defining the path URI

```
@PostMapping("/v1/bookNow")
@PostMapping("/v2/bookingNow")
```

2- Using @RequestParam (Request parameter versioning)

```
@RequestParam(name = "version") int version
```

3- Using @RequestHeader (name = "Api-Version") int version)

Q46. How to document REST API?

- Swagger API documentation (now called open api 3)

dependency required:

```
<dependency>
    <groupId> org.springdoc </groupId>
    <artifactId> springdoc-openapi-starter-webmvc-ui</artifactId>
    <version> 2.0.4 </version>
</dependency>
```

Everything else will be done by swagger itself.

Just open http://8080/swagger-ui/index.html

Use http://8080/v3/api-docs if you just want to share api descriptions

Q47- How to hide certain endpoints to prevent them from being exposed externally?

- @Hidden is used to hide any specific endpoint in documentation

Example:

```
@PostMapping("/v2/BookNow")
@Hidden
public ResponseEntity<String> bookTicket(@RequestBody Object obj){
        // implement logic
}
```

- @Operation annotation is used to define any custom description of endpoint.

```
@Operation(description = "This is test endpoint")
```

Q48- How to consume restful API?

1- RestTemplate
2- FeingClient
3- WebClient
4- Advanced Rest Client

1 RestTemplate Example

```
@Autowired
private RestTemplate template;

@GetMapping("/fetchMockUsers1")
public List<UserResponse> fetchUsingRestTemplate() {
    return template.getForObject("https://url.com/users", List.class);
}
```

2 Webclient example.

```
public List<UserResponse> fetchMockUsersWithWebClient() {
  Flux<UserResponse> response = webClient.get()
        uri("/users")
        .retrive()
        .bodyToFlux(UsersResponse.class);
  return response.collectList().block();
  }
}
```

Q 49- How will you handle exception in your product?

1- Use of @ControllerAdvice

- Create a global exception handler using @ControllerAdvice. This allows to define centralized way to handle exceptions thrown by any controller.

```
@ControllerAdvice
public class GlobalExceptionHandler {
  @ExceptionHandler(ResourceNotFoundException.class)
  public ResponseEntity<String> handleResourceException(ResourceNotFoundEx-e){

    return new ResponseEntity<>(ex.getMessage(), HttpStatus.NOT_FOUND);
  }

}
```

2- Custom Exception Class

```
public class ResourceNotFoundException extends RuntimeException{
    public ResourceNotFoundException(String message){
        super(message);
    }
}
```

3- Response Structure

```
public class ErrorResponse{
    private String message;
    private int status;
    private long timestamp;

    public ErrorResponse(String message, int status){
        this.message = message;
        this.status = status;
        this.timestamp = System.currentTimeMillis();
    }
    // Getters & Setters
}
```

4- Return Error Responses

```
@ExceptionHandler(ResourceNotFoundException.class)
public ResponseEntity<ErrorResponse> handleResourceExcpt(ResourceNotFoundException ex){

    ErrorResponse err = new ErrorResponse(ex.getMessage, HttpStatus.NOT_FOUND);
return new ResponseEntity<>(errorResponse, HttpStatus.NOT_FOUND);
}
```

5- Using Logger to log the errors

```
logger.error("Error Occured");
```

Import ~~org.slfj~~ org.slf4j.Logger for this.

Q50 How to avoid defining handlers for multiple exceptions, or what is the best practice for handling exceptions?

1- Use a base exception class
2- Generic exception handlers
3- Customizing Response with @ResponseStatus
@ResponseStatus(HttpStatus.NOT_FOUND)
4- Loggine exceptions
5- Testing your exception handlers

Q51- How will you validate or sanatize your input payload?
- import spring-boot-starter-validation in pom.xml

In your @RequestBody model class use on fields
@NotNull(message = "This field is required")
@NotEmpty(message = "This field can't be empty")
@Min(value = 10, message = " ")
@Max(value = 20, message = " ")
@NotBlank(message = "Cannot be null or empty")
@Pattern(regexp = "[A-Za-z0-9]+")
@Email
@DecimalMin
@DecimalMax
@AssertTrue // always true
@AssertFalse

```
@Future
private Date futureDate;
@Past
private Date pastDate;
```

After defining all these validation is model class use @Valid annotation along with @RequestBody in your controller.

MethodArgumentNotValid exception will be thrown by controller if you provide invalid inputs

Q 52- How can you populate validation error to end user?

-

```
@ExceptionHandler(MethodArgumentNotValidException.class)
public Map<String,String> handleError(MethodArgumentNotValidException ex) {

Map<String,String> errMap = new HashMap<>();
ex.getBindingResult().getFieldErrors().forEach(error->{
        errMap.put(error.getField(), error.getDefaultMessage());
});
return errMap;
}
}
```

Q5: Custom Bean Validation?

- Create Interface
ValidateProductType.java

```
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Constraint(validatedBy = ProductTypeValidator.class)
public @interface ValidateProductType{

public String message() default "Invalid productType"

Class<?>[] groups() default {};

Class<? extends Payload>[] payload() default {};
}
```

ProductTypeValidator.java

```
public class ProductTypeValidator implements ConstraintValidator<ValidateProductType, String>{

@Override
public boolean isValid(String productType, ConstraintValidatorContext constraintValidatorContext){
  List<String> productTypes = Arrays.asList("Electronics", "Education");

    return productTypes.contain(productType);
  }
}
```

Use the annotation to the field now :

```
@ValidProductType
private String productType;
```

Q54- You found bug in production environment, how will you debug it?

- First switch the environment from local to production. It can be done using spring profile. spring.profiles.active = prod or @Profile("prod")

Q55- Can we enable specific environment without using profiles? Or what is the alternative to profiles to achieving same use case?

- We can use condition instead of profiles to achieve same functionality

```
@ConditionalOnProperty(prefix = "app.active", name = "envi", havingValue = "dev")
```

It is just one example @Conditional provides many more annotation to specify env. or properties.

Q56 Difference between @Profile & @Conditional?

- @Profile or spring.profile.active is only for activating specific profile but Condition gives more feature for conditional bean creation and configuration in the app

Q57- What is AOP?

- AOP stands for Aspect Oriented Program and AOP provides a way to saparate cross-cutting concerns or secondary logic from the main business logic of our application.

Secondary logic could be transaction related code, validation, logging, auditing or notification related logic.

So these things are secondary to our code and without these also our code can run. So we should not mix these cross-cutting concerns and business concerns

Suppose I have a service where I want each method to have transation, logging, validation, auditing & notification. So, to not repeat any code for each method inside service we can create aspects for these features like transaction aspect or auditing aspect.

Join Point

Join point is the target method on which you want to apply your aspect.

Point Cut

The expression to tell to the AOP that who is your join point. Suppose you want aspect for saveProduct the you'll just define the pointcut for it.

Q-8- Different type of advice in AOP?

- Before Advice
- After Advice (don't care about Exception)
- After Returning Advice (execute if no exception)
- After throwing Advice (after exception is thrown)
- Around Advice - (execute before and after execution)

AOP Example

```
@Aspect
@Component
@Slf4j
public class LoggingAdvice{
  @PointCut("execution(* com.javatechie.service.ProductService.*(..))")
  private void logPointcut(){
  }

@Before("logPointcut()")
Public void logRequest(JoinPoint joinPoint) throws JsonProcessingException{
log.info("Before Advice")
}

// Similar execution for all other types of Advice

}
```

means for all methods

Q59- How application interact with database and which framework are you using?

- JDBC, Hibernate, JPA, Spring data JPA
But if specific to Spring boot then Spring data JPA is used.

In Spring boot app, the interaction with database involves using an Object Relational Mapping (ORM) framework like Hibernate (part of JPA - Java Persistence API) or Spring Data JPA. The framework simplifies the interaction with the database by allowing developers to perform CRUD opererations.

Steps to interact with database:

1- Data source configuration in application.properties or @Configuration.
2- Entity class (Models)
3- Repository layer - Provides built-in methods
4- Service layer

Q60- Why it is important to configure physical naming strategy?

- This strategy dictates how java entity field names are mapped to actual database column names.

Why it's important -

1- Consistency with database naming conventions: physical naming strategy ensures that the naming conventions followed in the database (such as snake-case) matkh the field names in the Java code.
2- Custom mapping facility: mapping field in java with a field in database with different names or case sensitivity.

3. Avoid Errors due to Case Sensitivity : A well-configured physical naming strategy helps avoid unexpected errors or mismatches caused by case differences.
4. Control over Naming Conflicts : Some of the database colums might have reserved keyword as their name so it can be controlled using naming strategy.

Example:

```
userAccount class
@Entity
@Table(name = "user_account")
public class userAccount {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String userName;
    private String accountNumber;

    // Getters & Setters
}
```

without naming strategy, Hibernate would attempt to map the java field userName directly to userName in the database, which may not match the user_name column in the database.

Configuring physical naming strategy in properties:

spring.jpa.hibernate.naming.physical-strategy = org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl

This tells Hibernate to use a standard physical naming strategy, which converts camelCase to snake_case.

Now userName will map to user_name
Now accountNumber will map to account_number

Q61- Key benefits of using Spring Data JPA.

- Spring Data JPA provides several key benefits that streamline the development of data access layers in Java applications, especially when working with relational databases.

1- Simplified Data Access
By automatic Repository generation, JPA provides ready-to-use CRUD operations through repository interface such as JpaRepository or CrudRepository. No need to write basic operations such as findAll(), save(), saveAll(), deleteById()

2- Custom Query Methods with Minimal Code
Derived Query methods: Spring Data JPA allows you to define query methods based on the method names. You can create complex queries by simply following a naming convention, which reduces the need to write

explicit JPQL or SQL queries.

Example:

```
List<Users> findByLastName(String lastName);
List<Users> findByAgeGreaterThan(int age);
```

3- Supports JPQL and Native Queries
Flexible Query Support: While Spring Data JPA offers derived query methods, it also allows you to write custom JPQL or native SQL queries when needed. This flexibility is crucial for handling complex queries.
~~Example~~ Example:

```
@Query("select u from users u where u.email = :email")
User findByEmail(@Param("email") String email);
```

4. Pagination and Sorting
Built-in Pagination and Sorting: Spring Data JPA provides easy-to-use pagination and sorting functionality out of the box. You can paginate or sort query results by simply passing Pageable or Sort parameters to repository methods.
Example:

```
Page<User> findAll(Pageable pageable);
List<User> finadAll(Sort sort);
```

5- Transaction Management

- Declarative Transaction Management : Spring Data JPA integrates seamlessly with Spring's transaction management, allowing you to use annotations like @Transactional to manage transactions declaratively. You don't need to manually manage transactions, which helps in keeping code clean and focused.

Transaction management means grouping multiple operations together. If one of them fails then nothing will change. Either all operation succeed or none of them take effect.

Benefits of Transaction Managements :

1- Automatic Rollback on failure.

2- Declarative Style (No manual handling required) : You don't need to write complex transaction management logic (like manually commiting or rolling back) using EntityManager or JDBC. Spring does this behind the scenes.

6- Abstraction over Persistence API

- Vendor Independencies : Spring Data JPA abstracts away the underlying persistence API, such as Hibernate or Eclipselink, allowing you to switch between JPA providers with minimal changes to your codebase.
- Reduced boilerplate : You don't need to write complex EntityManager or SessionFactory code manually, Spring Data JPA manages it for you.

7- Automatic Auditing
Audit fields: Spring Data JPA supports automatic auditing of entities using annotations like @CreatedDate, @LastModifiedDate, @CreatedBy and @LastModifiedBy. Useful for tracking creation and update times or users.

@CreatedDate and @LastModifiedDate Example:

```
@Entity
@EntityListener(AuditingEntityListener.class)
public class User {
 @CreatedDate
  private LocalDateTime createdAt;

 @LastModifiedDate
 private LocalDateTime updatedAt;

}
```

8- Optimistic locking: Define field version using @Version to ensure concurrency doesn't happen. It is a scenario where multipl users might update same data at same time and to avoid lost update problem.

9- Lazy or Eager loading:
Fetching strategies: Spring Data JPA provides control over how entity associations are fetched, using lazy or eager loading strategy. This helps

optimizing performance by controlling when relate data is fetched from the database.

Example:

```
@OneToMany(fetch = FetchType.LAZY)
private List<Book> books;

@ManyToOne (fetch = FetchType.EAGER)
private Publisher publisher;
```

10- Integration with other Spring Technologies
- Integrates well with other Spring modules such as Spring Security, Spring Batch, Spring REST, Spring MVC makes it easy to build full-stack applications.

Q62- Difference between Hibernate, JPA and Spring Data JPA?
- To work with Java Persistence API it is important to understand these frameworks:

1- Hibernate
Definition: Hibernate is a object-Relational mapping (ORM) framework for Java that provides a mechanism to map Java objects to database tables and vice versa.
Purpose: It simplifies database interactions by allowing developers to work with Java objects

instead of SQL queries.

Features:

- Session Management: Handles the lifecycle of entities and provides methods to create, read, update and delete objects.
- Caching: Supports first-level (session-level) and second-level caching for performance optimization.
- HQL: Hibernate Query Language allows you to write queries using entity object models insted of SQL.
- Custom dialect: Supports various database dialects to ensure compatibility with different relational databases.

2- Java Persistence API (JPA)

Definition: JPA is a specification for object-relational mapping in Java. It defines a set of APIs for managing relational data in java applications.

Purpose: Provides a standard interface and guidelines for ORM, which different implementations like Hibernate can follow.

Features:

- Annotations: Provides annotations like @Entity, @Table, @Id, etc. for mapping Java classes to database tables.
- Entity Manager: Manages the persistence context and transactions.
- Query Language: Defines JPQL (Java Persistence Query) for querying entities.
- Standardization: Promotes the use of a consistent

approach to ORM across different implementations, allowing for easier switching between different ORM frameworks.

3. Spring Data JPA

Definition: Spring Data JPA is a part of the Spring Data project that aims to simplify data access by providing a higher-level abstraction over JPA.

Purpose: Reduces boilerplate code and improves the ease of implementing data access layers in Spring applications.

Features:

- Repository Pattern: Provides CRUDRepository interface and JpaRepository interfaces to implement common CRUD operations without writing boilerplate code.
- Custom Queries: Allows custom query using @Query
- Pagination & Sorting: Built-in support for pagination and sorting.
- Integration with Spring: easy implementation of integration with transaction management of Spring.

Conclusion:

- Hibernate: ORM tool that implements the JPA specifications and provide additional features.
- JPA: is the standard ORM in Java, which can be implemented by various frameworks including Hibernate.
- Spring Data JPA: builds on top of JPA (and often

Hibernate) to simplify the data access layer, providing a more developer-friendly experience.

Feature	Hibernate	JPA	Spring Data JPA
Type	ORM framework	Specification	Data access framework
Level	Implementation	Abstraction	Abstraction on top of JPA
Boilerplate Reduction	Moderate	None	High
Repository Pattern	No	No	Yes
Custom queries	HQL / Criteria API	JPQL	Method name queries / @Query
Integration	Standalone or with JPA	with any ORM	Integrate with Spring

Q63- Connect multiple database or data source in a single application?

1- Define connection properties for both datasource

```
Spring.datasource.primary.url = jdbc:mysql://localhost:3306/primarydb
// other properties
Spring.datasource.secondary.url = jdbc:mysql://localhost:3306/secondarydb
```

2- Create configuration class for each data source. Mark one datasource as @Primary while others will be explicitly referenced when needed.

3- Ensure defining Entities and Repositories in separate packages so that configuration Bean for each datasource can find the desired packages.

Q64- Different ways to define custom queries in Spring Data JPA?

-① Derived Query Methods

Spring Data JPA allows defining query methods on the method name convention. These methods are already implemented by parsing the method name.

Example

```
List<User> findByLastName(String name);
```

② JPQL (Java Persistence Query Language) queries with @Query Annotation

```
@Query("select u from u where u.email = ?1")
User findByEmail(String email);

@Modifying
@Query("Update users u set u.name = "Vishal");
void updateName( );
```

③ Native queries with @Query

```
@Query(value = "Select * from users where email=?1", nativeQuery=true)
User findByEmailNative(String email);
```

Native queries can use datase-specif syntax and functions.

4- Named Queries
Defined at entity level using @NamedQuery or @NamedNativeQuery

```
@Entity
@NamedNativeQuery(name = "User.findByStatus", query = "  ")
public class User {

    // fields
}
```

In repository:

```
List<User> findByStatus(String status);
```

Q-65- How to define entity relationships or associations mapping in Spring Data JPA?

- entity relationships are defined using JPA annotations to map how entites (DB tables) relate to each other.

1- One-to-One
2- One-to-Many
3- Many-to-One
4- Many-to-Many

1- One-to-One Relationship
each entity is related to one instance of another.
Example:
User and Profile

User class

```
@Entity
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.Identity)
    private Long id;

    private String username;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name="profile_id", referencedColumnName = "id")
    private Profile profile;

    // Getter & Setters
}
```

Profile Class

```
@Entity
public class Profile {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String address;

    @OneToOne(mappedBy = "profile")
    private User user;
    // Getter & Setter
}
```

- Cascade = CascadeType.ALL indicates that operations such as remove are cascaded from User to Profile.

- @JoinColumn specifies the foreign key columns that joins the User table to the Profile table.

2. One-To-Many Relationship
Parent Child relationship but each child can be related to only one parent.
Example:
Department can have multiple Employees

```
Department Class
@Entity
public class Department {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  private String name;

  @OneToMany(mappedBy = "department", cascad = CascadeType.ALL)
  private List<Employee> employees;

  // Getters and Setters
}
```

Employee Class

```
@Entity
public class Employee{

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @ManyToOne
    @JoinColumn(name = "department_id", referencedColumnName = "id")
    private Department department;

    // Getter & Setter
}
```

- mappedBy = "department" specifies the field in the Employee entity that owns the relationship.
- @JoinColumn specifies the foreign-key column (department_id) on the Employee table.

3. Many-to-One Relationship

Many instances of one entity can be related to one instance of another entity.

In the above example just reverse the prespective then Many employee will have one department.

4- Many-To-Many Relationship

- each instance of one entity can be related to multiple instances of another entity and vice versa.

Example:
Student & Courses

:Student Class

```
@Entity
public class Student {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  private String name;

  @ManyToMany
  @JoinTable(name = "student_course", joinColumns = @JoinColumn(name =
                                                    "student_id"),
                                      inverseJoinColumns = @JoinColumn(name = "course_id"
  )
  private List<Course> courses;

    // Getters & Setters
}
```

Course Class

```
@Entity
public class Course {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String title;

    @ManyToMany(mappedBy = "courses")
    private List<Student> students;

    // Grefters Setters
}
```

- joinColums : Specifies the foreign key column that relates the student table to the join table.
- inverseJoinColumns : Specifies the foreign key column that relates the Course table to the join table.
- mappedBy = "courses" indicates that relationship is owned by the Student entity.

In microservice architecture the mapping is not recommended for loose coupling.

Q.66. It it possible to execute join query in spring data jpa ? If yes, how can you add some Insights?

- Yes it's possible to execute JOIN query in Spring Data JPA.
- You can use:

1- JPQL (Java Persistence Query Language)
2- Native SQL queries
3- Spring Data JPA @Query annotation

Q.67- Implement Pagination and Sorting in Spring Data JPA.

REPO

```
public interface EmployeeRepo extends JPARepository<Employee, Long>{

  Page<Employee> findAll(Pageable pageable);

}
```

SERVICE

```
@Service
public class EmployeeService{
  @Autowired
  private EmployeeRepo employeeRepo;

public Page<Employee> getEmployees(int age, int size, String sortBy){

PageRequest pageRequest = PageRequest.of(page, size, Sort.by(sortBy));

  return employeeRepo.findAll(pageRequest);
  }
}
```

- Entity of Employee is standard. No extra annotations required
- Controller method will take pageNo, size, sortBy in RequestParam.

Important Annotations in Spring Boot

1- Spring Boot Main Annotations
These annotations are fundamental to setting up and configuring a Spring Boot application, enabling features like auto-configuration and component scanning.
@SpringBootApplication: Combines @Configuration, @EnableAutoConfiguration and @ComponentScan to simplify Spring Boot application setup.
@ComponentScan: Specifies the packages to scan for Spring Components.
@EnableAutoConfiguration: Activates Spring Boot's auto-configuration capabilites based on the classpath.
@Configuration: Marks a class as a source of bean definitions for the Spring container.

2- Stereotype Annotations
Stereotype annotations define the role of a class in the Spring application context, facilitating dependency injection and component management.
@Component: Indicates that a class is a Spring-managed component.
@Service: Marks a class as a service provider in

the service layer.
@RestController : A specialized controller that handles RESTful requests and ~~request~~ returns data directly.
@Controller : Marks a class as a Spring MVC controller for handling web requests.
@Repository : Indicates that a class provides data access functionality, enabling exception ~~handling~~ translations.

3- Spring Core Related Annotations
These annotations are used for configuring beans, injecting dependencies and managing application properties.
@Configuration : Indicates a class that defines bean methods.
@Bean : Declares a method that produces a bean for the Spring container.
@Autowired : Automatically injects dependencies into a class.
@Qualifier : Specifies which bean to inject when multiple candidates exists.
@Lazy : indicates that a bean should be initialized lazily, upon first access. Spring delay the initialization until it's required.
@Value : Injects values from properties files or environment variables into fields.
@PropertySource : Specifies the location of external properties file.

@ConfigurationProperties: Binds properties from configuration files to a java object.
@Profile: Specifies which beans are eligible for registration based on active profiles.
@Scope: Defines the scope of a bean (singleton or prototype etc).

4- REST API Related Annotations

These annotations are specific to building RESTful web services, mapping HTTP requests to controller methods.
@RestController: Combines @Controller and @ResponseBody for RESTful web services.
@RequestMapping: Maps HTTP requests to handler methods.
@GetMapping: Maps HTTP Get requests to a specific handler method.
@PostMapping: Maps POST request.
@PutMapping: Maps PUT request.
@DeleteMapping: Maps DELETE request.
@RequestBody: Binds the body of an HTTP request to a method parameter.
@PathVariable: Extracts value from URI template variables.
@RequestParam: Binds query parameters to method parameters.
@ControllerAdvice: Global exception handling across controllers.
@ExceptionHandler: Handles specific exceptions thrown by controller methods.

5. Spring Data JPA Related Annotations.

These annotations are used for mapping Java objects to database tables and defining relationships between entities.

@Entity: Marks a class as a JPA entity.

@Table: Specifies database table name to which an entity is mapped.

@Column: Configures the details of a database column.

@Transactional: Indicates that a method should run within a transactional context.

Entity Class Relationships:

@OneToOne: one to one relationship between two entities.

@OneToMany: One to many relationship between entities.

@ManyToOne: Many to one relationship between entities

@ManyToMany: Many to Many relationship between entities.