Miscellaneous

1. IoC and Dependency Injection

IoC stands for Inversion of Control, which means the control of creating and managing objects is moved from the developer to the Spring framework. Instead of us creating objects using new keyword, Spring creates and manages them. Dependency Injection is the practical implementation of IoC. With Dependency Injection, Spring injects required dependencies into a class automatically. For example, if a service needs a repository, Spring creates the repository object and injects it into the service. This reduces tight coupling, makes the code easier to test, and improves maintainability.

Example

```
@Service

public class OrderService {

   private final OrderRepository repo;

   public OrderService(OrderRepository repo) {

      this.repo = repo;

   }

}
```

2. @Component vs @Service vs @Repository

@Component, @Service, and @Repository are all stereotype annotations used to mark Spring-managed beans. @Component is a generic annotation used for utility or helper classes. @Service is used for business logic classes and clearly represents the service layer. @Repository is used for data access classes and provides an additional benefit of exception translation, where database exceptions are converted into Spring's DataAccessException. Functionally they are similar, but they improve readability and layer separation, which is very important in real projects.

Example

```
@Component

public class DateUtil { }


@Service

public class UserService { }


@Repository

public class UserRepository { }
```

3. @Autowired vs @Qualifier vs @Primary

@Autowired is used to inject a dependency automatically by type. If only one bean of that type exists, Spring injects it without any issue. If multiple beans of the same type exist, Spring gets confused. In that case, @Qualifier is used to specify exactly which bean should be injected. @Primary is used to mark one

bean as the default choice when multiple beans are present.
Example

```
@Service

public class PaymentService {

   public PaymentService(@Qualifier("upiPayment") PaymentGateway gateway) {

   }

}
```

4. Bean scopes and bean lifecycle
Bean scope defines how many instances of a bean Spring creates. The default scope is singleton, where only one instance is created and shared across the application. Prototype scope creates a new instance every time the bean is requested. Other scopes like request and session are used in web applications. Bean lifecycle defines the stages a bean goes through, such as creation, dependency injection, initialization, usage, and destruction. Spring allows hooks at different stages using annotations like @PostConstruct and @PreDestroy.
Example

```
@Scope("prototype")

@Component

public class TempBean { }
```

5. Spring Boot auto-configuration internal working
Spring Boot auto-configuration works by detecting dependencies present in the classpath and automatically configuring beans based on them. Internally, it uses @EnableAutoConfiguration, which loads auto-configuration classes listed in spring.factories or AutoConfiguration.imports. These configuration classes use conditional annotations like @ConditionalOnClass and @ConditionalOnMissingBean. For example, if Spring Web is present and no DispatcherServlet bean is defined, Spring Boot creates one automatically. This mechanism reduces manual configuration and makes Spring Boot applications quick to develop.
Example

```
@ConditionalOnClass(DataSource.class)

public class DataSourceAutoConfig { }
```

7. How to disable auto-configuration
Sometimes Spring Boot auto-configures beans that we don't want. In such cases, we can disable auto-configuration explicitly. This can be done using the exclude attribute of @SpringBootApplication or @EnableAutoConfiguration. It tells Spring Boot not to apply specific auto-configuration classes. For example, if we don't want DataSource auto-configuration because we are not using a database, we can exclude it. This gives us fine-grained control while still using Spring Boot.
Example

```
@SpringBootApplication(exclude = DataSourceAutoConfiguration.class)

public class Application { }
```

8. DispatcherServlet request flow
DispatcherServlet is the front controller in Spring MVC. Every HTTP request first comes to DispatcherServlet. It then consults the HandlerMapping to find the correct controller method for the request. Once found, the request is passed to the controller. The controller processes the request and returns a response or a view name. DispatcherServlet then uses ViewResolver if needed and sends the final response back to the client. This centralized flow helps manage routing, validation, exception handling, and response generation in a clean way.
Example

```
@RestController

public class HelloController {

  @GetMapping("/hello")

  public String hello() {

    return "Hello";

  }

}
```

9. @Controller vs @RestController
@Controller is used in Spring MVC applications where the response is a view, such as JSP or Thymeleaf. It returns a logical view name and is mainly used for server-side rendered applications. @RestController, on the other hand, is used for REST APIs and returns data like JSON directly. Internally, @RestController is a combination of @Controller and @ResponseBody. In real projects, I use @Controller for UI-based applications and @RestController for APIs and microservices.
Example

```
@RestController

public class ApiController {

  @GetMapping("/api/data")

  public String data() {

    return "data";

  }

}
```

10. @RequestParam vs @PathVariable
@RequestParam is used to extract values from query parameters in the URL. It is commonly used for filtering, sorting, or optional parameters. @PathVariable is used to extract values from the URL path itself and usually represents a resource identifier. For example, user slash 10 clearly identifies a user, while user question mark role equals admin is a filter. In REST design, @PathVariable is preferred for mandatory identifiers and @RequestParam for optional data.
Example

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

```
public User getUser(@PathVariable int id, @RequestParam(required = false) String role) {
   return new User(id, "John");
}
```

11. Global exception handling using @ControllerAdvice

@ControllerAdvice is used to handle exceptions globally across all controllers in a Spring application. Instead of writing exception handling logic in every controller, we centralize it in one place. This improves code cleanliness and ensures consistent error responses. For example, if a NullPointerException or custom exception occurs in any controller, @ControllerAdvice can catch it and return a proper HTTP response with a meaningful message. This is very important in REST APIs where consistent error format is expected by clients.

Example

```
@ControllerAdvice
public class GlobalExceptionHandler {
   @ExceptionHandler(RuntimeException.class)
   public String handleException() {
      return "Something went wrong";
   }
}
```

12. @Bean vs @Component

@Bean and @Component are both used to create Spring-managed objects, but they are used in different situations. @Component is placed on the class itself and Spring detects it automatically through component scanning. @Bean is used on a method inside a @Configuration class and gives more control over object creation. I use @Component for my own classes and @Bean when configuring third-party classes like RestTemplate or ObjectMapper that I cannot modify.

Example

```
@Configuration
public class AppConfig {
   @Bean
   public RestTemplate restTemplate() {
      return new RestTemplate();
   }
}
```

13. application.properties vs application.yml

Both application.properties and application.yml are used to configure Spring Boot applications, and functionally they are the same. application.properties uses key-value format, while application.yml uses hierarchical and more readable structure. YML is preferred when configuration becomes large and grouped,

such as database or security settings. Properties file is simpler and commonly used for small configurations. Choice is mostly based on readability and team preference.
Example

```
server.port=8080

server:

  port: 8080
```

14. Profiles in Spring Boot and @Profile use case
Profiles are used to define environment-specific configurations like dev, test, and prod. @Profile allows us to load certain beans only when a specific profile is active. This is useful when different environments require different implementations, such as mock services in development and real services in production. Profiles avoid manual configuration changes during deployment and make applications environment-aware.
Example

```
@Profile("dev")

@Service

public class DevEmailService { }
```

15. CommandLineRunner vs ApplicationRunner
Both CommandLineRunner and ApplicationRunner are used to run code after the Spring Boot application starts. CommandLineRunner receives arguments as a simple String array, while ApplicationRunner provides structured access using ApplicationArguments. I use these interfaces for tasks like data initialization, loading cache, or startup validation. ApplicationRunner is preferred when arguments need to be parsed properly.
Example

```
@Component

public class StartupRunner implements CommandLineRunner {



    public void run(String... args) {

        System.out.println("App started");

    }

}
```

16. Embedded Tomcat working
Spring Boot comes with an embedded servlet container like Tomcat, which means we do not need to install or configure an external server separately. When the application starts, Spring Boot automatically creates and starts an embedded Tomcat instance inside the JVM. DispatcherServlet is registered with this Tomcat, and it begins listening on the configured port, usually 8080. When a request comes in, Tomcat receives it and forwards it to DispatcherServlet for processing. This approach makes applications self-contained, easy to deploy, and consistent across environments, because the server travels with the application.
Example

```
@SpringBootApplication

public class Application {

    public static void main(String[] args) {

        SpringApplication.run(Application.class, args);

    }

}
```

17. Circular dependency in Spring and solutions

Circular dependency occurs when two or more beans depend on each other directly or indirectly. For example, Service A depends on Service B, and Service B depends on Service A. Spring can resolve circular dependency only with setter injection, not with constructor injection. However, circular dependencies usually indicate a design issue. The best solution is to refactor the code, extract common logic into a third service, or use @Lazy to break the cycle. Constructor injection is preferred, so redesign is always the recommended fix.

Example

```
@Service

public class ServiceA {

    @Autowired

    @Lazy

    private ServiceB serviceB;

}
```

18. Testing REST controllers using MockMvc

MockMvc is used to test Spring REST controllers without starting the full server. It allows us to simulate HTTP requests and verify responses like status code, headers, and response body. This makes tests fast and focused only on controller logic. I use MockMvc with @WebMvcTest to load only the web layer, which avoids loading the entire application context. This is very useful for writing clean and efficient unit tests for APIs.

Example

```
@WebMvcTest(UserController.class)

public class UserControllerTest {

    @Autowired

    private MockMvc mockMvc;

}
```

19. Auto-generating custom primary key values in Spring Boot

By default, @Id with @GeneratedValue supports standard strategies like AUTO, IDENTITY, SEQUENCE, and TABLE, which generate numeric values. These strategies cannot generate custom patterns like ORD 1001 or

USER 2024 001. If we need a custom formatted primary key, we must handle it manually using application logic or database support.

The most common and recommended approach is to use a sequence for numeric generation and then add the custom prefix in code. The database generates a unique number, and the application formats it. This ensures uniqueness, performance, and avoids race conditions.

Example

```
@Entity

public class Order {


    @Id

    private String orderId;


    @PrePersist

    public void generateId() {

        this.orderId = "ORD-" + System.currentTimeMillis();

    }

}
```

20. How to connect a database in Spring Boot

First, we add the required dependencies. For example, if we are using a relational database like MySQL or PostgreSQL, we add spring-boot-starter-data-jpa and the corresponding database driver. Spring Boot uses these dependencies to understand which database it needs to configure.

Second, we configure the database properties in application.properties or application.yml. Here we define the database URL, username, password, and the JDBC driver class. Spring Boot reads these properties at startup and automatically creates a DataSource bean. We do not need to write any manual connection code.

Example

```
spring.datasource.url=jdbc:mysql://localhost:3306/testdb

spring.datasource.username=root

spring.datasource.password=secret

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true
```

Third, Spring Boot performs auto-configuration. Based on the presence of JPA and database dependencies, it automatically configures DataSource, EntityManagerFactory, and TransactionManager. This is handled internally using @EnableAutoConfiguration and conditional annotations, so no boilerplate configuration is needed.

Next, we define entity classes using @Entity and repositories using JpaRepository. Spring Boot connects these entities to the database tables automatically through Hibernate.

Example

```
@Entity

public class User {

   @Id

   @GeneratedValue

   private Long id;

   private String name;

}

public interface UserRepository extends JpaRepository<User, Long> {

}
```

At runtime, when the application starts, Spring Boot establishes the database connection, validates it, and manages it using a connection pool like HikariCP. Whenever a repository method is called, Spring fetches a connection from the pool, executes the query, and returns the connection safely.

21. Cyclic dependency

Cyclic dependency happens when two or more components depend on each other directly or indirectly, creating a loop. In simple terms, Class A depends on Class B, and Class B depends on Class A. Because of this cycle, the system cannot decide which object to create first, which leads to initialization problems.

In Java and Spring, cyclic dependency is most commonly discussed in the context of Spring beans. For example, if ServiceA has a dependency on ServiceB and ServiceB also has a dependency on ServiceA using constructor injection, Spring fails at startup with a circular dependency error. This is because Spring tries to create both objects at the same time and gets stuck.

Example

```
@Service

class ServiceA {

   private final ServiceB serviceB;

   public ServiceA(ServiceB serviceB) {

      this.serviceB = serviceB;

   }

}
```

```
@Service
class ServiceB {
    private final ServiceA serviceA;
    public ServiceB(ServiceA serviceA) {
        this.serviceA = serviceA;
    }
}
```

Spring can sometimes resolve cyclic dependency only with setter or field injection, because it can create one bean first and inject the dependency later. However, relying on this is not recommended. Cyclic dependency usually indicates a design problem.

The best solutions are to refactor the design, extract the common logic into a third service, or use @Lazy to break the cycle temporarily. Constructor injection is preferred in modern Spring, so redesigning the dependency structure is the cleanest solution.

22. Can you use multiple bean Configurations in Spring Boot?

Yes. Just have to add the paths to componentscan.

```
@ComponentScan(basePackages = {"com.yourapp", "com.other.configpackage"})
```

23. Aspect Oriented Programming AOP

Aspect Oriented Programming, or AOP, is a programming approach used to separate cross cutting concerns from business logic. Cross cutting concerns are functionalities that are needed across multiple parts of the application, such as logging, security, transaction management, performance monitoring, and auditing. If we write this logic inside every business method, the code becomes repetitive and hard to maintain. AOP solves this by allowing us to write such logic in one place and apply it across the application.

In Spring AOP, the core idea is that business logic should focus only on what it needs to do, and extra behavior should be added automatically at runtime. This is achieved using aspects and proxies. An aspect is a class that contains cross cutting logic. A join point is a point in the execution flow, such as method execution. A pointcut defines where the aspect should be applied. Advice defines what action should be taken, such as before a method runs, after it runs, or around it.

For example, instead of writing logging code inside every service method, we can write one logging aspect that logs method entry and exit automatically. Spring creates a proxy object around the target class, and whenever a method is called, the proxy executes the advice at the defined pointcut.

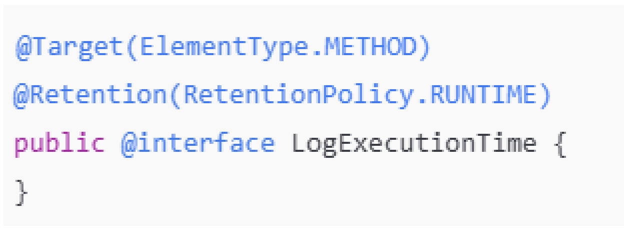
Example

```
@Aspect
@Component
public class LoggingAspect {
    @Before("execution(* com.app.service.*.*(..))")
    public void logBefore() {
        System.out.println("Method called");
    }
}
```

In real projects, AOP is heavily used by Spring internally. For example, @Transactional uses AOP to start and commit or roll back transactions around method execution. Similarly, security checks and performance monitoring are also implemented using AOP.

24. How to define custom annotation in spring boot?
You can define a custom annotation in Spring Boot when you want to apply reusable logic across multiple classes or methods, such as logging, validation, or security. You define it using @interface, and then use AOP or custom logic to handle its behavior at runtime.

eg.



```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface LogExecutionTime {
}
```

25. Can I replace @SpringBootApplication with @Configuration, @EnableAutoConfiguration, and @ComponentScan
Yes, you can replace @SpringBootApplication with @Configuration, @EnableAutoConfiguration, and @ComponentScan because @SpringBootApplication is actually a convenience annotation that internally combines these three. Functionally, both approaches are equivalent. @Configuration marks the class as a configuration class, @EnableAutoConfiguration tells Spring Boot to auto configure beans based on classpath dependencies, and @ComponentScan tells Spring where to scan for components. In real projects, we use @SpringBootApplication because it is cleaner, less error prone, and is the standard convention. Replacing it manually is mostly done for learning or special customization, not in day to day production code.

26. Can I replace @RestController with @Controller
Yes, you can replace @RestController with @Controller, but you must explicitly add @ResponseBody to each handler method. @RestController is a shortcut that combines @Controller and @ResponseBody. If you use only @Controller without @ResponseBody, Spring will treat the return value as a view name instead of JSON. In REST APIs, @RestController is preferred because it avoids repetition and clearly indicates that the controller is meant for REST responses. Using @Controller is more suitable for MVC applications that return HTML views like Thymeleaf.

27. Can I replace @Controller, @Service, and @Repository with @Component
Yes, technically you can replace @Controller, @Service, and @Repository with @Component because all of them are specializations of @Component. Spring will still detect them as beans during component

scanning. However, in real applications this is not recommended. @Service and @Repository add semantic meaning to the code and help readability. @Repository also enables exception translation for database related exceptions, and @Controller is used by Spring MVC to identify request handling components.

i.e.

For a **@Service class**, replacing it with @Component usually works without any visible issue. The class will still be detected as a Spring bean and dependency injection will work normally. However, you lose the semantic meaning that this class contains business logic. There is no functional break, but readability and design clarity suffer. No code changes are required, but best practice is still to keep @Service for business layer classes.

For a **@Repository class**, replacing it with @Component can cause real behavioral changes. @Repository enables Spring's exception translation mechanism, which converts low-level persistence exceptions into Spring's DataAccessException hierarchy. If you replace @Repository with @Component, this automatic exception translation will not happen. If your code or global exception handling depends on DataAccessException, behavior can change. To compensate, you would need to handle persistence exceptions manually, which is why replacing @Repository with @Component is strongly discouraged.

For a **@Controller class**, replacing it with @Component breaks request handling unless you add extra annotations. Spring MVC detects controllers only through @Controller or @RestController. If you use only @Component, Spring will create the bean, but it will not register request mappings. To make it work, you must explicitly add @RequestMapping annotations and also add @ResponseBody if you want JSON responses. This makes the code messy and defeats the purpose of using Spring MVC annotations.

Annotations

1. What is @SpringBootApplication?
@SpringBootApplication is the main annotation used to start a Spring Boot application. Internally, it combines @Configuration, @EnableAutoConfiguration, and @ComponentScan. This means the class can define beans, Spring Boot will automatically configure components based on dependencies, and it will scan packages for Spring beans. Because of this single annotation, we don't need to write separate configuration code. For example, if Spring Web dependency is present, DispatcherServlet is configured automatically without any manual setup. This annotation mainly exists to reduce boilerplate and make application startup faster.
Example

```
@SpringBootApplication

public class DemoApplication {

   public static void main(String[] args) {

      SpringApplication.run(DemoApplication.class, args);

   }

}
```

2. What is @ComponentScan?
@ComponentScan tells Spring where to search for Spring-managed beans. By default, Spring scans the package where the main class is located and all its sub-packages. If controllers or services are present outside this structure, Spring will not detect them unless we explicitly specify the base packages. I usually use @ComponentScan when my project has shared modules or common packages outside the main package.
Example

```
@ComponentScan(basePackages = "com.app.common")

@SpringBootApplication

public class DemoApplication { }
```

3. What is @EnableAutoConfiguration?
@EnableAutoConfiguration tells Spring Boot to automatically configure beans based on the dependencies present in the classpath. For example, if Spring Data JPA is available, it configures DataSource and EntityManager automatically. If Spring Web is present, it configures DispatcherServlet. This works using conditional logic and avoids manual configuration. It is one of the core reasons Spring Boot is easy to use.
Example

```
@EnableAutoConfiguration

public class AppConfig { }
```

4. What is @Configuration?
@Configuration marks a class as a source of bean definitions. It replaces XML-based configuration. Methods inside this class annotated with @Bean return objects that are managed by the Spring container. I use @Configuration mainly when I need to configure third-party libraries or create custom beans manually.
Example

```
@Configuration

public class AppConfig {

   @Bean

   public RestTemplate restTemplate() {

      return new RestTemplate();

   }

}
```

5. What is @Component?
@Component is a generic annotation used to mark a class as a Spring-managed bean. It is mainly used for helper classes, utility classes, or components that do not clearly belong to service or repository layers. Once annotated, Spring automatically creates and injects this class wherever required.
Example

```
@Component

public class DateUtils { }
```

6. What is @Service?
@Service is used to mark business logic classes. It improves readability by clearly separating business logic from other layers. Spring also uses this annotation to apply service-level features like transaction management. For example, order processing or payment logic is typically written inside a service class.
Example

```
@Service

public class OrderService {

   public void placeOrder() { }

}
```

7. What is @Controller?
@Controller is used in Spring MVC applications where the response is a view such as JSP or Thymeleaf. It handles HTTP requests and returns a logical view name instead of JSON. I use @Controller when building traditional server-side rendered web applications.
Example

```
@Controller

public class PageController {

   @GetMapping("/home")

   public String home() {

      return "home";

   }

}
```

8. What is @RestController?
@RestController is used to build REST APIs and returns JSON or XML directly. Internally, it combines @Controller and @ResponseBody, so we don’t need to add @ResponseBody on every method. I use @RestController when building APIs for frontend, mobile, or microservices communication.
Example

```
@RestController

@RequestMapping("/users")

public class UserController {

   @GetMapping("/{id}")

   public User getUser(@PathVariable int id) {

      return new User(id, "John");

   }

}
```

9. What is @Repository?
@Repository is used for data access layer classes that interact with the database. It is commonly used with JPA or JDBC. One important feature is exception translation, where database-specific exceptions are converted into Spring’s DataAccessException, making error handling consistent.
Example

```
@Repository

public class UserRepository {

  public User findById(int id) {

    return new User(id, "Alex");

  }

}
```

10. What is @Bean?
@Bean is used to define a Spring-managed object inside a configuration class. It is especially useful when working with third-party classes that cannot be annotated directly. Spring manages the lifecycle of the object returned by the method and injects it wherever required.
Example

```
@Configuration

public class BeanConfig {

  @Bean

  public ObjectMapper objectMapper() {

    return new ObjectMapper();

  }

}
```

11. What is @Autowired?
@Autowired is used to inject dependencies automatically by type. It tells Spring to find a matching bean from the application context and inject it into the required class. This removes the need for manual object creation and tight coupling. In real projects, I prefer constructor injection with @Autowired because it makes dependencies mandatory, improves testability, and avoids null-related issues. Field injection works but is not recommended for production code.
Example

```
@Service

public class OrderService {

  private final UserRepository userRepository;

  @Autowired

  public OrderService(UserRepository userRepository) {
        this.userRepository = userRepository;

    }

}
```

12. What is @Qualifier?
@Qualifier is used along with @Autowired when more than one bean of the same type exists. In such cases, Spring cannot decide which bean to inject and throws an error. @Qualifier explicitly tells Spring which bean to use. This is very common when multiple implementations of an interface are present.
Example

```
@Service

public class PaymentService {

    @Autowired

    public PaymentService(@Qualifier("upiPayment") PaymentGateway gateway) {

    }

}
```

13. What is @Lazy?
@Lazy tells Spring to create a bean only when it is actually needed, instead of creating it at application startup. By default, Spring creates all singleton beans eagerly. @Lazy is useful when a bean is heavy, slow to initialize, or rarely used. This helps improve application startup time and reduce memory usage.
Example

```
@Lazy

@Service

public class ReportService {

}
```

14. What is @Value?
@Value is used to inject values from properties files into variables. It helps externalize configuration and avoids hardcoding values inside code. This is commonly used for injecting URLs, credentials, limits, or timeout values. It also makes applications easier to configure across environments.
Example

```
@Component

public class AppConfig {

    @Value("${app.timeout}")

    private int timeout;

}
```

15. What is @PropertySource?
@PropertySource is used to load an external or custom properties file into the Spring environment. It is

useful when configuration is stored outside application.properties or application.yml. This allows better separation of configuration and code.
Example

```
@PropertySource("classpath:custom.properties")

@Configuration

public class PropertyConfig {

}
```

16. What is @ConfigurationProperties?
@ConfigurationProperties binds multiple related configuration values into a single Java class. It is cleaner and more maintainable than using many @Value annotations. This is commonly used for grouping database, mail, or security configurations. It also improves readability and reduces configuration errors.
Example

```
@Component

@ConfigurationProperties(prefix = "db")

public class DatabaseConfig {

   private String url;

   private String username;

}
```

17. What is @Profile?
@Profile is used to activate beans only for specific environments such as dev, test, or prod. This helps load environment-specific implementations without changing code. For example, using a mock service in development and a real service in production.
Example

```
@Profile("dev")

@Service

public class DevEmailService {

}
```

18. What is @Scope?
@Scope defines the lifecycle and visibility of a Spring bean. The default scope is singleton, which means only one instance is created. Prototype scope creates a new instance every time it is requested. Prototype is useful when state should not be shared between users or requests.
Example

```
@Scope("prototype")

@Component

public class TempObject {
}
```

19. What is @RequestMapping?
@RequestMapping is used to map HTTP requests to controller methods. It can be applied at class level or method level and can define URL paths and HTTP methods. It acts as a base mapping for other specific mappings like GetMapping and PostMapping.
Example

```
@RestController

@RequestMapping("/api/users")

public class UserController {

}
```

20. What is @GetMapping?
@GetMapping is used to handle HTTP GET requests. It is mainly used to fetch data from the server. It is a shortcut for @RequestMapping with method set to GET. In REST APIs, it is commonly used for read operations.
Example

```
@GetMapping("/{id}")

public User getUser(@PathVariable int id) {

   return new User(id, "Sam");

}
```

21. What is @PostMapping?
@PostMapping is used to handle HTTP POST requests in Spring REST APIs. It is mainly used when we want to create or submit new data to the server. In RESTful design, POST is used for create operations. It is a shortcut for @RequestMapping with method set to POST and makes the code more readable. For example, when creating a new user or placing an order, @PostMapping is used to accept request data from the client.
Example

```
@PostMapping("/users")

public User createUser(@RequestBody User user) {

   return user;

}
```

22. What is @PutMapping?
@PutMapping is used to handle HTTP PUT requests and is mainly used to update existing resources. In REST APIs, PUT is idempotent, meaning calling it multiple times produces the same result. I use @PutMapping when updating user details or modifying existing records. It clearly indicates update intent and improves API readability.
Example

```
@PutMapping("/users/{id}")
```

```
public User updateUser(@PathVariable int id, @RequestBody User user) {
    return user;
}
```

23. What is @DeleteMapping?
@DeleteMapping is used to handle HTTP DELETE requests. It is used when we want to remove a resource from the system. This makes the API semantics clear and follows REST principles. For example, deleting a user or removing an order uses @DeleteMapping.
Example

```
@DeleteMapping("/users/{id}")
public void deleteUser(@PathVariable int id) {
}
```

24. What is @RequestBody?
@RequestBody is used to map incoming JSON data from the HTTP request body into a Java object. It is commonly used in POST and PUT APIs. Spring automatically converts JSON into Java objects using message converters like Jackson. This avoids manual parsing and makes API development easier.
Example

```
@PostMapping("/orders")
public Order createOrder(@RequestBody Order order) {
    return order;
}
```

25. What is @PathVariable?
@PathVariable is used to extract values directly from the URL path. It is useful when the value is part of the resource identifier. For example, user slash 10 clearly identifies a user with id 10. This makes URLs more RESTful and readable.
Example

```
@GetMapping("/users/{id}")
public User getUser(@PathVariable int id) {
    return new User(id, "Alex");
}
```

26. What is @RequestParam?
@RequestParam is used to extract values from query parameters in the URL. It is used when the parameter is optional or used for filtering and searching. For example, searching users by role or status uses query parameters.
Example

```
@GetMapping("/users")
public List<User> getUsers(@RequestParam String role) {
    return List.of();

}
```

27. What is @ControllerAdvice?
@ControllerAdvice is used for global exception handling across all controllers. It allows us to centralize error handling logic and return consistent error responses. This avoids repeating exception handling code in every controller and improves maintainability.
Example

```
@ControllerAdvice

public class GlobalExceptionHandler {

}
```

28. What is @ExceptionHandler?
@ExceptionHandler is used to handle specific exceptions inside a controller or a ControllerAdvice class. It allows us to customize error messages and HTTP responses for different exceptions. This gives full control over error handling behavior.
Example

```
@ExceptionHandler(RuntimeException.class)

public String handleException() {

    return "Error occurred";

}
```

29. What is @Entity?
@Entity marks a class as a JPA entity that is mapped to a database table. Each instance of the entity represents a row in the table. It is the core concept of ORM, where Java objects are mapped to database records automatically.
Example

```
@Entity

public class User {

    @Id

    private int id;

}
```

30. What is @Table?
@Table is used to specify table details such as table name or schema. It is useful when the table name does not match the entity class name or when additional table-level settings are required.
Example

```
@Entity

@Table(name = "users")

public class User {
}
```

31. What is @Column?
@Column is used to customize how a class field is mapped to a database column. By default, JPA maps field names directly to column names, but @Column allows us to define properties like column name, length, nullable, and uniqueness. This helps enforce database constraints from the application level itself. For example, marking a column as non-null ensures invalid data is not stored and keeps database design aligned with business rules.
Example

```
@Entity

public class User {

  @Id

  private int id;

  @Column(nullable = false, unique = true)

  private String email;

}
```

32. What is @Transactional?
@Transactional ensures that a method runs inside a database transaction. All database operations inside the method either complete successfully together or fail together. If an exception occurs, Spring automatically rolls back the transaction. I usually use @Transactional at the service layer because business logic often spans multiple database operations. This ensures data consistency, especially in operations like money transfer or order processing.
Example

```
@Service

public class AccountService {

  @Transactional

  public void transferMoney() {

    debit();

    credit();

  }

}
```

33. What is @OneToOne?
@OneToOne defines a one-to-one relationship between two entities, meaning one record in a table is associated with exactly one record in another table. A common example is a user and user profile, where each user has only one profile. This relationship helps model real-world constraints clearly in the database and application.
Example

```
@Entity

public class User {

   @OneToOne

   private Profile profile;

}
```

34. What is @OneToMany?
@OneToMany defines a relationship where one parent entity is associated with multiple child entities. For example, one customer can have multiple orders. This is commonly used in parent-child table relationships. It helps retrieve related data easily while maintaining proper normalization in the database.
Example

```
@Entity

public class Customer {

   @OneToMany

   private List<Order> orders;

}
```

35. What is @ManyToOne?
@ManyToOne represents the inverse side of a one-to-many relationship. Many child records are linked to one parent record. For example, many orders belong to one customer. This annotation is very common in relational database mappings and helps define foreign key relationships clearly.
Example

```
@Entity

public class Order {

   @ManyToOne

   private Customer customer;

}
```

36. What is @ManyToMany?
@ManyToMany defines a relationship where multiple records from one table are associated with multiple records from another table. A common example is users and roles, where one user can have multiple roles and one role can belong to multiple users. This relationship is implemented using a join table internally.
Example

```
@Entity

public class User {

   @ManyToMany

   private List<Role> roles;

}
```

Hibernate

1. What is JPA vs Hibernate?
JPA is a specification, while Hibernate is an implementation of that specification. JPA defines a set of rules and interfaces for object relational mapping, but it does not provide actual code. Hibernate provides the actual implementation and functionality. In simple terms, JPA tells what should be done, and Hibernate tells how it is done. In real projects, we code using JPA annotations and interfaces so that our application is not tightly coupled to Hibernate. Hibernate also provides extra features like caching and performance optimizations beyond JPA.
Example

```
@Entity

public class User {

   @Id

   private Long id;

}
```

2. Entity vs DTO vs POJO
A POJO is a simple Java object with no special restriction. An Entity is a POJO that is mapped to a database table using JPA annotations. A DTO is used only to transfer data between layers or systems and should not contain business logic or persistence annotations. In real applications, entities are used for database operations, DTOs are used for API responses, and POJOs are the general building blocks. Separating these helps avoid exposing database structure directly to clients.
Example

```
@Entity

public class UserEntity { }



public class UserDTO { }
```

3. FetchType LAZY vs EAGER
FetchType defines when related data is loaded from the database. LAZY means related data is loaded only when it is accessed, while EAGER means related data is loaded immediately along with the parent entity. LAZY improves performance by avoiding unnecessary database queries, while EAGER can cause performance issues if large data is loaded unnecessarily. In real projects, LAZY is preferred, and EAGER is used only when data is always required.
Example

```
@OneToMany(fetch = FetchType.LAZY)

private List<Order> orders;
```

4. Why @OneToMany default is LAZY?
@OneToMany default fetch type is LAZY because loading child collections eagerly can be very expensive. A

parent entity may have hundreds or thousands of child records, and loading all of them every time can slow down the application. By keeping it LAZY, Hibernate loads the child data only when it is actually needed. This design choice helps prevent performance and memory issues in production systems.
Example

```
@OneToMany

private List<Order> orders;
```

5. N plus 1 select problem and JOIN FETCH
The N plus 1 select problem happens when Hibernate executes one query to fetch parent entities and then executes additional queries for each parent to fetch related data. This results in many database calls and poor performance. JOIN FETCH solves this by fetching parent and child data in a single query using a join. This reduces database calls and improves performance.
Example

```
@Query("SELECT u FROM User u JOIN FETCH u.orders")

List<User> findUsersWithOrders();
```

6. Cascade types and orphanRemoval
Cascade types define how operations on a parent entity are propagated to its child entities. Common cascade types are PERSIST, MERGE, REMOVE, REFRESH, and ALL. For example, if CascadeType.PERSIST is used, saving the parent will automatically save the child. orphanRemoval is slightly different. It means if a child entity is removed from the parent's collection, Hibernate will delete that child record from the database. Cascade handles parent-driven operations, while orphanRemoval handles child lifecycle cleanup. In real projects, orphanRemoval is useful to avoid stale or unused child records.
Example

```
@OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)

private List<Order> orders;
```

7. Entity states in Hibernate
Hibernate entities go through different states during their lifecycle. Transient state means the object is created but not associated with any session and not stored in the database. Persistent state means the object is associated with a Hibernate session and any changes are tracked automatically. Detached state means the session is closed, but the object still exists in memory. Removed state means the entity is marked for deletion. Understanding these states is important to avoid unexpected database behavior.
Example

```
User user = new User(); // Transient

entityManager.persist(user); // Persistent
```

8. Dirty checking
Dirty checking is a Hibernate feature where it automatically detects changes made to persistent entities and updates the database without explicitly calling update. Hibernate keeps a snapshot of the entity state when it is loaded and compares it before committing the transaction. If any field is changed, Hibernate generates the update query automatically. This reduces boilerplate code and ensures consistency. Dirty

checking works only when the entity is in persistent state.
Example

```
@Transactional

public void updateName(User user) {

   user.setName("New Name");

}
```

9. Level 1 vs Level 2 cache
Level 1 cache is the first-level cache associated with a Hibernate session. It is enabled by default and exists for the lifetime of the session. It prevents repeated database queries within the same session. Level 2 cache is optional and shared across sessions. It stores frequently accessed data to reduce database calls. Level 2 cache improves performance but must be used carefully.
Example

```
@Cacheable

@Entity

public class Product { }
```

10. What should not be cached
Not all data should be cached. Frequently changing data, sensitive information like passwords, transactional data, and real-time data should not be cached. Caching such data can lead to stale results, security risks, or data inconsistency. In real systems, only stable, read-heavy data like reference or master data should be cached. This ensures correctness while still gaining performance benefits.

11. @Version and optimistic locking
@Version is used to implement optimistic locking in JPA and Hibernate. Optimistic locking assumes that conflicts are rare and does not lock database rows. Instead, it uses a version column to detect concurrent updates. Every time an entity is updated, the version value is incremented. If two transactions try to update the same record, the second transaction fails because the version has changed. This prevents lost updates and ensures data consistency without heavy locking. Optimistic locking is widely used in high-concurrency systems because it performs better than pessimistic locking.
Example

```
@Entity

public class Product {

   @Id

   private Long id;

   @Version

   private int version;

}
```

12. save vs saveAndFlush vs persist
save and saveAndFlush are Spring Data JPA methods, while persist is a JPA standard method. save stores

the entity in the persistence context but may delay writing to the database until flush happens. saveAndFlush immediately writes changes to the database by forcing a flush. persist only makes the entity managed and does not return the saved entity. In real applications, save is used most of the time, saveAndFlush is used when we need data immediately in the database, and persist is used when sticking strictly to JPA specification.
Example

```
repository.save(entity);

repository.saveAndFlush(entity);

entityManager.persist(entity);
```

13. get() vs load()
get and load are Hibernate methods used to fetch entities. get fetches the data immediately and returns null if the record does not exist. load returns a proxy object and hits the database only when the data is accessed. If the record does not exist, load throws an exception. In real projects, get is safer because it avoids unexpected exceptions, while load is useful when we are sure the record exists and want lazy loading behavior.
Example

```
User user = session.get(User.class, 1L);

User proxy = session.load(User.class, 1L);
```

14. Methods of JpaRepository
JpaRepository provides many ready-made methods for database operations, which reduces boilerplate code. Common methods include save, findById, findAll, deleteById, existsById, count, and flush. It also supports pagination and sorting. In real projects, JpaRepository allows developers to focus on business logic instead of writing repetitive CRUD code. Custom query methods can also be created by following naming conventions.
Example

```
public interface UserRepository extends JpaRepository<User, Long> {

   List<User> findByName(String name);

}
```

<u>Security</u>

1. Authentication vs Authorization
Authentication and authorization are two different but related security concepts. Authentication is about verifying who the user is, while authorization is about checking what the user is allowed to do. Authentication happens first, usually using username and password or a token. Once the user is authenticated, authorization checks roles or permissions before allowing access to resources. For example, logging into an application is authentication, but accessing an admin page is authorization. Even if a user is authenticated, they cannot access resources without proper authorization.
Example

```
http.authorizeHttpRequests(auth -> auth

   .requestMatchers("/admin").hasRole("ADMIN")

   .anyRequest().authenticated()

);
```

2. How does SecurityFilterChain work in Spring Boot 3+?
In Spring Boot 3 and above, security is configured using SecurityFilterChain instead of WebSecurityConfigurerAdapter. SecurityFilterChain defines a chain of security filters that every HTTP request passes through. These filters handle authentication, authorization, CSRF, CORS, and session management. When a request comes in, it flows through the filters in order, and each filter performs its responsibility before the request reaches the controller. This approach is more explicit and flexible and follows modern Spring design principles.
Example

```
@Bean

public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

   http.authorizeHttpRequests(auth -> auth.anyRequest().authenticated())

      .httpBasic();

   return http.build();

}
```

3. UserDetailsService and implementation
UserDetailsService is a core Spring Security interface used to load user-specific data during authentication. When a user tries to log in, Spring Security calls the loadUserByUsername method to fetch user details such as username, password, and roles. This data is then used to authenticate the user. In real applications, UserDetailsService is implemented to fetch user data from a database. This allows Spring Security to remain decoupled from how users are stored.
Example

```
@Service

public class CustomUserDetailsService implements UserDetailsService {

   public UserDetails loadUserByUsername(String username) {

      return User.withUsername(username)

            .password("password")

            .roles("USER")

            .build();

   }

}
```

4. JWT-based authentication flow
JWT-based authentication is a stateless authentication mechanism. First, the user logs in with credentials. If authentication is successful, the server generates a JWT token and sends it to the client. The client then sends this token in the Authorization header with every request. On each request, the server validates the token and extracts user details from it. Since no session is stored on the server, this approach scales well and is widely used in microservices and REST APIs.
Example

```
Authorization: Bearer eyJhbGciOiJIUzI1NiJ9
```

5. Role-based and method-level security
Role-based security restricts access to endpoints based on user roles like USER or ADMIN. Method-level security provides finer control by securing individual methods using annotations. This is useful when security rules are complex and cannot be handled only at URL level. Method-level security is commonly enabled using @EnableMethodSecurity.
Example

```
@PreAuthorize("hasRole('ADMIN')")

public void deleteUser() { }
```

6. Custom login and logout flows
Spring Security allows customizing login and logout behavior instead of using default pages. We can define custom login endpoints, success handlers, and logout URLs. This is useful when integrating with frontend applications or building custom UI. Logout configuration ensures tokens or sessions are properly cleared.
Example

```
http.formLogin(login -> login.loginPage("/login"))

    .logout(logout -> logout.logoutUrl("/logout"));
```

7. Securing REST APIs best practices
Securing REST APIs requires using stateless authentication like JWT, enforcing HTTPS, validating input, and applying proper authorization checks. CSRF is usually disabled for stateless APIs. Roles should be checked both at endpoint and method levels. Sensitive data should never be exposed, and proper HTTP status codes should be returned. Logging and rate limiting are also important to prevent abuse.
Example

```
http.csrf(csrf -> csrf.disable())

    .authorizeHttpRequests(auth -> auth.anyRequest().authenticated());
```

8. JWT vs OAuth2
JWT and OAuth2 are often confused, but they solve different problems. JWT is a token format, while OAuth2 is an authorization framework. JWT stands for JSON Web Token and is a compact, self-contained token that carries user information like username, roles, and expiry. It is commonly used for stateless authentication. For example, after login, the server issues a JWT, and the client sends this token with every request. The server validates the token without storing any session, which makes JWT fast and scalable.

OAuth2, on the other hand, is a complete authorization framework that defines how a client can access a protected resource on behalf of a user. It is commonly used when third-party applications need access, like allowing a mobile app to log in using Google or GitHub. OAuth2 does not mandate JWT, but JWT is often used as the access token format in OAuth2 implementations.

REST API

1. HTTP Methods:

GET
GET is used to retrieve data from the server. It does not modify any data and should be safe and idempotent. GET requests can be cached and bookmarked. In real applications, GET is used to fetch user details, lists, or reports. Sensitive data should not be sent in GET parameters because they appear in URLs.
Example

GET /users/10

POST
POST is used to create a new resource on the server. It is not idempotent, meaning calling it multiple times may create multiple records. POST requests send data in the request body and are commonly used for form submission or creating entities like users or orders.
Example

POST /users

PUT
PUT is used to update or replace an existing resource. It is idempotent, so calling it multiple times with the same data gives the same result. PUT usually updates the entire resource.

Example

PUT /users/10

PATCH
PATCH is used to partially update a resource. Unlike PUT, it updates only specific fields instead of replacing the whole object. This is useful when only a few fields change, such as updating user status or email. PATCH is not always supported by all clients but is common in modern APIs.
Example

PATCH /users/10

DELETE
DELETE is used to remove a resource from the server. It is idempotent, meaning deleting the same resource multiple times gives the same result. In real systems, DELETE may perform soft deletes instead of physical removal.
Example

DELETE /users/10

HEAD
HEAD is similar to GET but returns only headers, not the body. It is used to check whether a resource exists or to validate cache headers without transferring data.
Example

HEAD /users/10

OPTIONS
OPTIONS is used to discover which HTTP methods are supported by a server or endpoint. It is commonly used in CORS preflight requests by browsers.
Example

```
OPTIONS /users
```

2. What are HTTP status codes and why are they important?
Status codes tell the client the result of the request. 200 means success, 201 means created, 400 means bad request, 401 unauthorized, 403 forbidden, 404 not found, and 500 server error. Proper status codes make APIs predictable and easier to debug.

3. What is idempotency?
Idempotency means calling an API multiple times gives the same result. GET, PUT, and DELETE are idempotent, while POST is not. This is important for retry mechanisms in distributed systems.

4. PUT vs PATCH

PUT and PATCH are both used to update resources in a REST API, but they are used in different situations. PUT is used to update or replace an entire resource. When we send a PUT request, the server expects the full representation of the resource. If some fields are missing, they may be overwritten or reset. PUT is idempotent, which means calling it multiple times with the same data will always produce the same result. In real applications, PUT is used when the client has the complete object and wants to fully update it.

PATCH is used for partial updates. Instead of sending the full object, we send only the fields that need to be changed. This reduces payload size and avoids accidental overwriting of unchanged fields. PATCH is not always idempotent, because the result may depend on how the update is applied. PATCH is commonly used for small updates like changing status, email, or flags.

Example

```
PUT /users/10

{

  "id": 10,

  "name": "John",

  "email": "john@example.com"

}

PATCH /users/10

{

  "email": "newjohn@example.com"

}
```