1. What is Dependency Injection (DI) in Spring? Explain different types of DI.

Answer:

Dependency Injection (DI) is a core concept in Spring that allows for the injection of dependencies into an object at runtime rather than compile time. This promotes loose coupling and improves testability.

Types of DI in Spring:

Constructor Injection: Dependencies are injected through a class constructor. It’s immutable and recommended for mandatory dependencies.

Setter Injection: Dependencies are injected through setter methods. It provides flexibility for optional dependencies but can be modified after object creation.

Field Injection: This type injects dependencies directly into fields using @Autowired annotations on private fields. Although concise, it’s generally discouraged in favor of constructor injection for testing ease and immutability.

Example:

// Constructor Injection

public class Service {

private final Repository repository;

@Autowired

public Service(Repository repository) {

this.repository = repository;

}

}

2. What is Spring Boot, and how does it simplify application development?

Answer:

Spring Boot is an extension of the Spring framework that simplifies the development of stand-alone, production-ready applications. It removes much of the boilerplate configuration by providing a variety of Starter dependencies and auto-configuration features, allowing developers to start coding with minimal setup.

Key features include:

Auto-Configuration: Automatically configures beans based on the application classpath and existing beans.

Spring Boot Starters: Predefined configurations for dependencies, such as spring-boot-starter-web for web applications.

Embedded Servers: Supports embedded servers like Tomcat, Jetty, and Undertow, eliminating the need for external server configuration.

3. How does Spring Boot Auto-Configuration work?

Answer:

Spring Boot’s auto-configuration automatically configures beans that might be needed by the application. It uses the @EnableAutoConfiguration annotation, which scans for specific classes in the classpath and automatically configures them as beans.

Process: The spring.factories file lists all configuration classes to be loaded. During startup, Spring Boot checks the application context to see if specific beans already exist; if not, it auto-configures them.

Customization: Auto-configuration can be customized using application.properties or application.yml, and specific configurations can be excluded by specifying them in @SpringBootApplication(exclude = {DataSourceAutoConfiguration.class}).

4. What is the purpose of @SpringBootApplication annotation?

Answer:

@SpringBootApplication is a composite annotation that combines three annotations:

@EnableAutoConfiguration: Enables Spring Boot’s auto-configuration mechanism.

@ComponentScan: Scans the package where the main class is located for Spring components like @Controller, @Service, etc.

@Configuration: Allows the class to be a source of bean definitions.

Example:

@SpringBootApplication

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

5. How does Spring Security work, and what are some common configurations?

Answer:

Spring Security is a powerful framework for authentication and authorization in Java applications. It provides comprehensive security services with features like role-based access control, CSRF protection, and OAuth integration.

Basic Components:

Authentication: Verifies the identity of a user or system.

Authorization: Defines what resources or actions an authenticated user can access.

Filters: Spring Security applies a series of filters that intercept HTTP requests to apply security logic.

Common Configurations:

In-Memory Authentication: Useful for testing, storing credentials in-memory.

Database Authentication: Configures Spring Security to authenticate users from a database.

Method-Level Security: Using annotations like @Secured or @PreAuthorize.

Example:

@Configuration

@EnableWebSecurity

public class SecurityConfig extends WebSecurityConfigurerAdapter {

@Override

protected void configure(HttpSecurity http) throws Exception {

http.authorizeRequests()

.antMatchers("/public/\*\*").permitAll()

.anyRequest().authenticated()

.and().formLogin().and().httpBasic();

}

}

6. What are Spring Profiles and how do you use them?

Answer:

Spring Profiles allow you to register different beans for different environments (e.g., development, test, production) and isolate environment-specific configurations.

Usage: Profiles can be defined using @Profile annotation on beans or specifying properties in application-dev.properties, application-prod.properties, etc.

Activation: Profiles can be activated through:

application.properties: spring.profiles.active=dev

Command Line: java -jar app.jar --spring.profiles.active=prod

7. Explain Spring AOP and its main components.

Answer:

Spring AOP (Aspect-Oriented Programming) is used for adding cross-cutting concerns like logging, transactions, and security without modifying the business logic.

Key Components:

Aspect: Modularizes a cross-cutting concern. Defined using @Aspect.

Advice: Code to be executed at a join point (before, after, around). Types include @Before, @After, @AfterReturning, and @Around.

Pointcut: Defines where the advice should be applied.

Join Point: A point in application execution (method call) where advice can be applied.

Example:

@Aspect

@Component

public class LoggingAspect {

@Before("execution(\* com.example.service.\*.\*(..))")

public void logBeforeMethod(JoinPoint joinPoint) {

System.out.println("Executing: " + joinPoint.getSignature().getName());

}

}

8. How does Spring Transaction Management work?

Answer:

Spring provides a consistent programming model for transaction management across different data sources and APIs.

Declarative Transaction Management: Achieved through @Transactional annotation. It simplifies transaction management by wrapping the annotated method in a transactional context.

Programmatic Transaction Management: Using TransactionTemplate or PlatformTransactionManager to control transactions manually.

Example of @Transactional:

@Transactional

public void saveOrder(Order order) {

orderRepository.save(order);

}

Propagation Levels: Defines how transactions behave in nested methods (e.g., REQUIRED, REQUIRES\_NEW, MANDATORY).

9. What is the use of @RestController in Spring Boot? How is it different from @Controller?

Answer:

@RestController is a specialized version of @Controller in Spring, intended for RESTful web services. It combines @Controller and @ResponseBody, meaning all methods return JSON or XML data directly.

@Controller is typically used in MVC applications for handling web pages. It returns a ModelAndView object and uses view resolvers to render HTML pages.

10. How does Spring Boot handle exceptions using @ControllerAdvice?

Answer:

@ControllerAdvice is a global exception handler that can be applied to multiple controllers. It provides a centralized exception handling mechanism.

Usage: Annotate a class with @ControllerAdvice and define methods with @ExceptionHandler to specify custom handling for exceptions.

Example:

@ControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(ResourceNotFoundException.class)

public ResponseEntity<Object> handleResourceNotFound(ResourceNotFoundException ex) {

return new ResponseEntity<>(ex.getMessage(), HttpStatus.NOT\_FOUND);

}

}

11. Explain the difference between @Component, @Service, @Repository, and @Controller in Spring.

Answer:

These annotations are all part of Spring’s stereotype annotations, used to indicate different roles of beans in a Spring application.

@Component: A generic stereotype for any Spring-managed component. It serves as a base annotation and can be used to denote any Spring bean.

@Service: A specialization of @Component, used specifically for service layer beans. It is commonly used to indicate business logic and service classes.

@Repository: A specialization of @Component, used for data access layers (DAOs). It provides automatic translation of database exceptions into Spring’s DataAccessException.

@Controller: A specialization of @Component, used in MVC applications. It handles HTTP requests and returns responses, often in combination with @RequestMapping.

Each annotation helps define the structure of the application by categorizing beans, which improves code readability and maintainability.

12. What is the difference between @RequestMapping and @GetMapping, @PostMapping, etc.?

Answer:

@RequestMapping is a versatile annotation in Spring MVC that maps HTTP requests to handler methods. It can be used for multiple HTTP methods (GET, POST, PUT, DELETE) by specifying the method attribute.

Example of @RequestMapping:

@RequestMapping(value = "/users", method = RequestMethod.GET)

public ResponseEntity<List<User>> getAllUsers() { ... }

@GetMapping, @PostMapping, etc. are shorthand annotations introduced in Spring 4.3 to simplify mappings for specific HTTP methods, making code more readable and concise.

Example of @GetMapping:

@GetMapping("/users")

public ResponseEntity<List<User>> getAllUsers() { ... }

13. What is a Spring Bean Lifecycle, and what are the common phases?

Answer:

Spring Bean Lifecycle involves various stages from creation to destruction. When a Spring container manages a bean, it goes through these main phases:

1. Instantiation: Spring creates an instance of the bean.

2. Dependency Injection: Spring injects dependencies as specified.

3. Post-Initialization (@PostConstruct): Any method annotated with @PostConstruct is called after dependencies are injected.

4. Initialization Callbacks (InitializingBean / @Bean(initMethod)): The afterPropertiesSet() method is called if the bean implements InitializingBean, or a custom init method can be specified.

5. Destruction (@PreDestroy, DisposableBean, or @Bean(destroyMethod)): When the container shuts down, methods annotated with @PreDestroy or destroy() from DisposableBean are called to release resources.

Example of Initialization and Destruction Callbacks:

@Bean(initMethod = "init", destroyMethod = "cleanup")

public MyBean myBean() {

return new MyBean();

}

14. Explain the @Configuration and @Bean annotations in Spring. How do they work together?

Answer:

@Configuration: Indicates that a class contains Spring bean definitions. The class is processed by Spring to generate beans in the application context.

@Bean: Used within a @Configuration-annotated class to define and register individual beans.

Together, they allow manual configuration of beans without using XML. Each @Bean method returns a bean instance, which is managed by the Spring IoC container.

Example:

@Configuration

public class AppConfig {

@Bean

public MyService myService() {

return new MyServiceImpl();

}

}

15. What is the difference between @Autowired and @Qualifier annotations in Spring?

Answer:

@Autowired: Used to inject dependencies automatically by type. It scans the Spring container for a matching type and injects it into the target bean.

@Qualifier: Used in conjunction with @Autowired when there are multiple beans of the same type to specify which bean should be injected. @Qualifier specifies the bean by name, which resolves the ambiguity.

Example:

@Autowired

@Qualifier("mySpecialService")

private MyService myService;

16. What is Spring Data JPA, and how does it simplify data access?

Answer:

Spring Data JPA is a part of the Spring Data project that simplifies data access by providing a higher-level API over JPA. It reduces the boilerplate code by offering a repository abstraction, which automatically generates common data access methods at runtime.

Key Features:

CrudRepository and JpaRepository: Predefined interfaces for CRUD operations.

Custom Query Methods: Methods are created based on method name patterns (e.g., findByLastName).

Query Annotation Support: Custom queries can be created using @Query.

Example of Repository:

public interface UserRepository extends JpaRepository<User, Long> {

List<User> findByLastName(String lastName);

}

17. What is the purpose of the @Transactional annotation in Spring? How does it work internally?

Answer:

@Transactional is an annotation used to manage transactions declaratively in Spring. It ensures that the method or class it annotates will be executed within a transactional context, meaning all database operations within it are part of a single transaction.

Rollback: Transactions can be rolled back automatically on certain exceptions, like RuntimeException or custom exceptions specified by the developer.

Propagation and Isolation Levels: @Transactional supports multiple propagation levels (e.g., REQUIRED, REQUIRES\_NEW) and isolation levels (e.g., READ\_COMMITTED).

Example:

@Transactional

public void updateAccount(Account account) {

accountRepository.save(account);

}

Internally, Spring uses AOP (Aspect-Oriented Programming) to manage transactions, intercepting the method calls and applying transactional behavior.

18. Explain the difference between @RequestBody and @ResponseBody in Spring MVC.

Answer:

@RequestBody: Used on a method parameter in a controller to bind the HTTP request body to that parameter. Typically used with POST and PUT requests to parse incoming JSON or XML data into a Java object.

@ResponseBody: Used on a method in a controller to indicate that the return value should be written directly to the HTTP response body as JSON or XML, rather than rendering a view.

Example:

@PostMapping("/user")

public ResponseEntity<User> addUser(@RequestBody User user) {

userService.save(user);

return ResponseEntity.ok(user);

}

19. How does Spring Boot support externalized configuration?

Answer:

Spring Boot allows configuration properties to be stored externally, enabling easy modification without needing to rebuild the application. The most common files are application.properties or application.yml.

Configuration Hierarchy: Spring Boot loads configuration properties from multiple sources in a specific order:

Application properties located in src/main/resources.

Environment variables and command-line arguments.

External files specified with SPRING\_CONFIG\_LOCATION.

This allows for environment-specific configurations and easy management of sensitive information.

20. What is the purpose of Spring Cloud?

Answer:

Spring Cloud provides tools for building distributed systems, microservices, and cloud-native applications. It is built on top of Spring Boot and offers a variety of tools for service discovery, configuration management, load balancing, and circuit breaking.

Key Modules:

Spring Cloud Config: Manages configuration properties for distributed applications.

Eureka: Provides service discovery.

Ribbon and Feign: Tools for client-side load balancing.

Hystrix: A circuit breaker for handling service failures gracefully.

Example Usage of Spring Cloud Config:

spring:

config:

import: "optional:configserver:http://localhost:8888"